University of South Alabama

JagWorks@USA

Theses and Dissertations

Graduate School

5-2022

Adversarial Machine Learning for the Protection of Legitimate Software

Colby Parker

University of South Alabama, cbp1222@jagmail.southalabama.edu

Follow this and additional works at: https://jagworks.southalabama.edu/theses_diss

Part of the Information Security Commons, and the Theory and Algorithms Commons

Recommended Citation

Parker, Colby, "Adversarial Machine Learning for the Protection of Legitimate Software" (2022). *Theses and Dissertations*. 35.
https://jagworks.southalabama.edu/theses_diss/35

This Dissertation is brought to you for free and open access by the Graduate School at JagWorks@USA. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of JagWorks@USA. For more information, please contact jherrmann@southalabama.edu.

ADVERSARIAL MACHINE LEARNING FOR THE PROTECTION OF LEGITIMATE SOFTWARE

A Dissertation

Submitted to the Graduate Faculty of the
University of South Alabama
in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

in

Computing

by
Colby B. Parker
B. S., University of South Alabama, 2017
M.S., University of South Alabama, 2018
May 2022

ACKNOWLEDGMENTS

First, I thank my advisor Dr. J. Todd McDonald. Without him, none of this would have been possible and I wouldn't have been able to come this far in my academic career. His support and teachings through the years helped to prepare me for this moment. He's always provided me with opportunities to grow and succeed and I truly appreciate him for being with me these past six years.

I thank my committee members Dr. Ryan Benton, Dr. Aviv Segev, Dr. Armin Straub, and Dr. Yuan Gu for their patience, their choice to be in my committee, and their feedback on my research. In addition, I'd like to thank Dr. Dimitrios Damopolous for his assistance and ideas.

I'm very thankful for my fiancé Asia Griffin for all of her support during this long process. She did her absolute best to encourage me and get me back on my feet when I was overwhelmed with it all. She made even the worst times so much better. I also thank my friends Cody Johnston, Jamie Howell, and Josh Gray for celebrating my successes and pushing me to be my best.

Lastly, I thank Dr. Todd Andel and Ms. Angela Clark for their management of the SFS program at South Alabama. The opportunities and support from them, both inside and out of the program, were above and beyond.

TABLE OF CONTENTS

Page

LIST OF TABLES viii

LIST OF FIGURES x

LIST OF ABBREVIATIONS xii

ABSTRACT xiii

CHAPTER I INTRODUCTION 1

1.1 Research Questions 6
1.2 Research Goals and Contributions 6
1.3 Document Outline 7

CHAPTER II BACKGROUND 8

2.1 Obfuscation 8

2.1.1 Obfuscation Transformations 11

2.1.1.1 Layout Transformations, ... 11
2.1.1.2 Data Transformations 12
2.1.1.3 Control Transformations 13

2.1.1.3.1 Virtualization 14
2.1.1.3.2 Just in Time Compilation 15
2.1.1.3.3 Control Flow Flattening 15
2.1.1.3.4 Opaque Predicates 16
2.1.1.3.5 Encoding Arithmetic. 17

2.1.2 Metrics for Obfuscation 17

2.2 Machine Learning 20

2.2.1 Categories 20

2.2.1.1 Supervised Learning 21
2.2.1.2 Unsupervised Learning, ... 21
2.2.1.3 Reinforcement Learning, ... 21

2.2.2 Phases.. 22

2.2.2.1 Data Collection 23
2.2.2.2 Training and Testing 23

2.3 Adversarial Machine Learning 24

2.3.1 Machine Learning Attack Surface 25
2.3.2 Techniques 27

2.3.2.1 Training... 28
2.3.2.2 Inference 28

2.3.3 Adversary Knowledge 29
2.3.4 Evasion Attacks 31

2.3.4.1 Whitebox Evasion 33
2.3.4.2 Blackbox Evasion. 34

2.3.5 Crafting Adversarial Examples.. 35

2.3.5.1 Deep Neural Network. 35
2.3.5.2 Fast Gradient Sign Method. 37
2.3.5.3 Carlini and Wagner Method.. 38

CHAPTER III 39

RELATED WORK 39

3.1 Metadata Recovery from Obfuscated Programs Using Machine Learning 39
3.2 ByteWise: A case study in neural network obfuscation identification 42
3.3 Fine-Grained Static Detection of Obfuscation Transforms Using Ensemble-Learning and Semantic Reasoning.. 45

CHAPTER IV METHODOLOGY 47

4.1 Dataset creation.. 48

4.1.1 Dataset source 48
4.1.2 Obfuscations 49

4.2 Detection Suite.. 51

4.2.1 Gadget Based Detection 52
4.2.2 Image Based Detection 53

4.3 Evasion of Detectors 54

4.3.1 Adversarial Obfuscation 54
4.3.2 Obfuscation Expansion 55
4.3.3 Impact on Obfuscation 57

4.4 Automation and Comparison 57

CHAPTER V OBFUSCATION CLASSIFICATION 59

5.1 Introduction 59
5.2 Background 61

5.2.1 Convolutional Neural Networks 61

5.3 Methodology 62

5.3.1 Dataset creation 62

5.3.1.1 Image Creation. 63
5.3.1.2 Disassembly. 63
5.3.1.3 Gadget Extraction. 64

5.3.2 Classifiers 65

5.3.2.1 CNN. 65
5.3.2.2 FCNN. 65
5.3.2.3 Opcode. 68
5.3.2.4 Gadgets. 69

5.4 Results 69

5.4.1 CNN 70
5.4.2 FCNN 70
5.4.3 Opcode 70
5.4.4 Gadgets 73

5.4 Discussion 74
5.5 Future Work 75
5.6 Conclusion 76

CHAPTER VI EVADING OBFUSCATION CLASSIFICATION 78

6.1 Adversarial Machine Learning 78

6.1.1 Introduction 78
6.1.2 Executable Adversarial Examples 79
6.1.3 Methodology 81

6.1.3.1 Distance Comparisons. 82
6.1.3.2 Adversarial Obfuscation. 83

6.1.3.2.1 Generating adversarial examples 83
6.1.3.2.2 Semantic nops 85

6.1.3.3 Using on Each Classifier. 85

6.1.4 Results 87
6.1.5 Discussion 87
6.1.6 Future Work 89
6.1.7 Conclusion 89

6.2 Opcode Expansion 90

6.2.1 Introduction 90
6.2.2 Methodology 91

6.2.3.1 Code Segments. 92
6.2.3.2 Uniform Expansion. 93
6.2.3.3 Profile Expansion. 94
6.2.3.4 Guided Expansion. 95

6.2.3 Results 96

6.2.3.1 Opcode Profiles. 96
6.2.3.2 Uniform. 96
6.2.3.3. Profile. 98
6.2.3.4 Guided. 99

6.2.4 Discussion 100
6.2.5 Future Work 101
6.2.6 Conclusion 102

CHAPTER VII IMPLEMENTATION AND EVALUATION 103

7.1 Introduction 103
7.2 LOKI Obfuscator Framework 104

7.2.1 Capabilities 105

7.2.1.1 Guided Obfuscation. 105
7.2.1.2 Adversarial. 105

7.2.1.3 Obfuscation Analysis. 106

7.3 Methodology 106

7.3.1 Adversarial Expansion 106
7.3.2 Metrics 108

7.3.2.1 Stealth. 108
7.3.2.2 Potency. 108
7.3.2.3 Resilience. 108
7.3.2.4 Cost. 109

7.4 Results 109

7.4.1 Impact on Stealth 109
7.4.2 Impact on Potency 109
7.4.3 Impact on Resilience 111
7.4.4 Impact on Cost 112

7.5 Discussion 112
7.6 Future Work & Conclusion 112

REFERENCES 114

BIOGRAPHICAL SKETCH 131

LIST OF TABLES

Table Page

1. Description of Classifiers. 51

2. F1-scores of CNN model at differing layers. 70

3. Classification f1-scores for fully convolutional model. 71

4. Naive Bayes single layer results. 71

5. Naive Bayes multi-layer results. 71

6. Decision Tree single layer results. 72

7. Decision Tree multi-layer results. 72

8. SVM single layer results. 72

9. SVM multi-layer results. 72

10. Naive Bayes single layer results. 73

11. Naive Bayes multi-layer results. 73

12. Decision Tree single layer results. 73

13. Decision Tree multi-layer results. 74

14. SVM single layer results. 74

15. SVM multi-layer results. 74

16. Semantic NOPs. 85

17. Classification results for Adversarial Obfuscation. 88

18. Results for Targeted attacks. 88

19. Feature amount and top 50 features. 97

20. Average instruction group counts and percentage increase. 98

21. Classification results for uniform expansion. 98

22. Classification results for profile expansion. 99

23. Classification results for guided expansion. 99

24. Guided expansion classification results. 100

25. Classification scores for our generated samples. 110

26. Cyclomatic complexity measures for transformation combinations. 110

27. Sample Size Comparison. 111

LIST OF FIGURES

Figure Page

1. An adversarial image using noise to fool detection. 5

2. Inlining and Outlining of transformations [6]. 13

3. Splitting one function into two [5]. 13

4. Control Flow Flattening [5]. 15

5. Opaque predicate used to introduce bogus control flow [17]. 16

6. Diagram of a generic ML system.. 22

7. ML pipeline with labels attack surfaces... 26

8. Known attacks and attack surfaces on ML systems [49]. 26

9. This figure shows the components of an ML system. 31

10. Four examples of adversarial inputs. 32

11. Example of a DNN.. 36

12. Flowchart showing side effects in code. 40

13. ML pipeline for [14]. 41

14. Classification accuracies for experiments 1 and 2 (in red). 42

15. Classification accuracies for experiments 1 and 2 (red). 42

16. Structure of the RNN model used for the paper [17]. 43

17. Results of BCF detection [17].. 44

18. Design steps for detection system [18]. 46

19. Sample program generated by Tigress. 49

20. Partial view of the script use to produce obfuscated variants. 50

21. Example of Gadget list from a binary. 52

22. Two examples of a binary converted to a Grayscale image. 53

23. Adversarial Obfuscation Process. 55

24. Overview of Code Expansion Process. 56

25. Outline of Methodology. 62

26.Sample assembly output. 64

27. Figure showing CNN architecture. 66

28.Fully Convolutional Neural Network Architecture. 67

29. Basic nop insertion. 80

30. Adversarial Obfuscation overview. 81

31. Waterfall version of Adversarial Obfuscation. 84

32. Architecture of CNN used to generate AEs. 86

33. Outline of expansion process. 93

34. Adversarial Expansion Process. 107

LIST OF ABBREVIATIONS

MATE	Man-at-the-End
ML	Machine Learning
AML	Adversarial Machine Learning
AE	Adversarial Example
DNN	Deep Neural Network
FGSM	Fast Gradient Sign Method
CW	Carlini-Wagner
TF-IDF	Term Frequency Inverse Document Frequency
CNN	Convolutional Neural Network
FCNN	Fully Convolutional Neural Network
SVM	Support Vector Machine
AO	Adversarial Obfuscation

ABSTRACT

Parker, Colby B., Ph.D, University of South Alabama, May 2022. Adversarial Machine Learning for the Protection of Legitimate Software. Chair of Committee: Jeffrey Todd McDonald, Ph.D.

Obfuscation is the transforming a given program into one that is syntactically different but semantically equivalent. This new obfuscated program now has its code and/or data changed so that they are hidden and difficult for attackers to understand. Obfuscation is an important security tool and used to defend against reverse engineering.

When applied to a program, different transformations can be observed to exhibit differing degrees of complexity and changes to the program. Recent work has shown, by studying these side effects, one can associate patterns with different transformations. By taking this into account and attempting to profile these unique side effects, it is possible to create a classifier using machine learning which can analyze transformed software and identifies what transformation was used to put it in its current state. This has the effect of weakening the security of obfuscating transformations used to protect legitimate software.

In this research, we explore options to increase the robustness of obfuscation against attackers who utilize machine learning, particular those who use it to identify the type of obfuscation being employed. To accomplish this, we segment our research into three stages. For the first stage, we implement a suite of classifiers that are used to

identify the obfuscation used in samples. These establish a baseline for determining the effectiveness of our proposed defenses and make use of three varied feature sets.

For the second stage, we explore methods to evade detection by the classifiers. To accomplish this, attacks setup using the principles of adversarial machine learning are carried out as evasion attacks. These attacks take an obfuscated program and make subtle changes to various aspects that will cause it to be mislabeled by the classifiers. The changes made to the programs affect features looked at by our classifiers, focusing mainly on the number and distribution of opcodes within the program. A constraint of these changes is that the program remains semantically unchanged. In addition, we explore a means of algorithmic dead code insertion in to achieve comparable results against a broader range of classifiers.

In the third stage, we combine our attack strategies and evaluate the effect of our changes on the strength of obfuscating transformations. We also propose a framework to implement and automate these and other measures. We the following contributions:

1. An evaluation of the effectiveness of supervised learning models at labeling obfuscated transformations. We create these models using three unique feature sets: Code Images, Opcode N-grams, and Gadgets.
2. Demonstration of two approaches to algorithmic dummy code insertion designed to improve the stealth of obfuscating transformations against machine learning: Adversarial Obfuscation and Opcode Expansion
3. A unified version of our two defenses capable of achieving effectiveness against a broad range of classifiers, while also demonstrating its impact on obfuscation metrics.

CHAPTER I

INTRODUCTION

A crucial component in the world economy that has grown immensely since the turn of the millennia is the software development industry. A broad global market and communications have been developed by dramatic improvements in technological functionality, internet and computer hardware efficiency, resulting in the convergence of computers and software in every area of human life: entertainment, education, military usage, medicine, transport [1]. This improvement and rapid expansion has had a large economic impact as well, with the software industry contributing more than $1 trillion to the United States economy and adding millions of jobs across a wide range of industries and at multiple skill levels [2]. This rapid growth and success have made the software industry a tempting target for crime and theft. In 2018, it was reported that cybercrime had taken potentially $109 billion from the U.S. economy in 2016 [3].

It should be no surprise that piracy is an issue the industry takes extremely seriously as intellectual property forms the backbone of its success. Laws in the United States define and protect intellectual property using suite of different classifications, all of which can be applied to different aspects of the software produced and sold by the industry [4]. Novel concepts, ideas, and features implemented in new software can be patented, allowing the developers holding the patent to obtain a competitive advantage

[4]. Companies worried about software being released that is eerily like theirs or that is their product repackaged can seek a copyright, which protects the way they have implemented their ideas [4]. While patents and copyrights can be infringed, trade secrets can be stolen if the company does not do their due diligence in keeping the secret safe [4]. Obtaining these classifications for their software allow players in the industry legal recourse in case someone tries to steal their ideas from them, and people and companies do try [4].

To preemptively defend their software from attack, developers and companies turn to software protection [5, 6, 7]. Protections in software take the form of modifications made to the original program that make it more resilient to attacks made by malicious actors [5]. A prominent form of software protection employed by developers at all levels is software obfuscation. Obfuscation is the process of taking a program and applying a transformative function in order to produce a new program that, while functionally equivalent to the original, now has code and/or data that is more concealed and is harder to understand for both people and automated systems. Obfuscation is a vital tool to security and is used heavily by companies and other entities to defend against reverse engineering of their created software.

The importance of obfuscation can be seen by examining a man-at-the-end (MATE) attack scenario. In this attack, the malicious actor has full access to the software and the machine running the software, with the developer of the software having no input as the actor is a legitimate user [8]. This is the scenario companies face when combating software piracy and tampering. An unprotected piece of software in this scenario could be easily reverse engineered, analyzed, and then be at the mercy of the bad actor. This

highlights the importance of obfuscation. Since obfuscation is placed into the code at development, it would be present in the MATE scenario and give the program some defense against malicious user [5-7].

In 2018, unlicensed, or pirated, software accounted for 37% of all software installed globally [9]. The money lost from this can have a strong impact on the global economy by hurting the software industry's ability to foster and promote job growth while also pushing innovation [10]. In a worst-case scenario, compromised intellectual property can affect not just the economy but extend its harm to national security. Since this is a threat faced by all companies great and small, some of which could not handle to financial impact of piracy, many companies seek to prevent it in the first place [11]. This leads more and more companies to make use of software protections such as obfuscation.

While obfuscating software may prevent an inexperienced adversary from gaining access to the software, it is not a silver bullet for protecting all software. With enough time and resources at their disposal, a sufficiently capable actor can eventually work through the transformations that have been applied to a piece of software and return it to something close enough to original form [7]. Due to the nature of MATE attacks, the attacker would always win in the end if nothing was done. Developers counteract this by using obfuscation not as an ironclad defense but as a way to buy time [6]. The goal becomes to make it so by the time the software has been reverse engineered it is no longer a victory for the attacker, either by making it so the software is obsolete or no longer functioning due to other factors [12].

Deobfuscation is the process of reverse engineering a piece of obfuscated software and retrieving or recreating the original program code. Although manual

deobfuscation is exceptionally reliable, it is a time-consuming process. Automatic deobfuscation tools are meant to speed up the process by automatically extracting some information or undoing an obfuscated transformation entirely [13, 14, 15]. These tools enable reverse engineers to scale their efforts with an increasingly large number of programs. There is a downside, as different obfuscating transformations often require specific techniques to undo and using these techniques on incorrect transformations would only make the process worse [13, 16]. Therefore, reverse engineers must first determine the type(s) of obfuscation(s) that were used on a given program to use any automated tools that they may possess. Identifying used obfuscations is often itself a manual task and therefore only increases the time needed to deobfuscate a program.

Obfuscations can be identified based on the unique changes different transformations will perform on the program as well as the impacts on the program's complexity [6, 14]. Recent work has shown, that by studying these side effects, one can associate patterns with different transformations via machine learning [14, 17, 18]. Searching for and extracting those patterns within the obfuscated a program's code or behavior enables machine learning classifiers to be trained to successfully identify the types of transformations used on obfuscated pieces of software. While this benefits the work of malware analysis, this also weakens the effectiveness of obfuscation used for protecting legitimate software. It may be possible as well to modify or remove the features created by obfuscation transformations, thus defeating the machine learning based identifiers.

The altering of features to evade machine learning models is part of the field of adversarial machine learning [19]. Attacking a machine learning model involves

analyzing the way the model makes decisions and using what is learned to slowly modify a sample in subtle ways to craft an adversarial sample [19]. These samples will be viewed by the model not as what they are but as what the attacker wants them to be. Figure 1 provides a visual example of an adversarial image crafted to be mislabeled by an image recognition model. Adversarial ML has been applied to a wide variety of field beyond just images, such as audio, health data, spam email, and even malware [19].

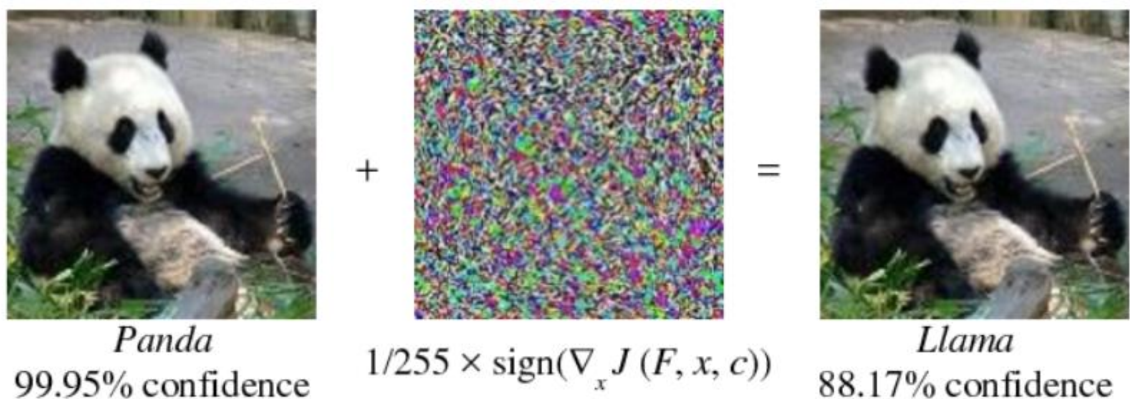

Figure 1. An adversarial image using noise to fool detection.

In this research, we examine if Adversarial ML can be leveraged for defense instead of attack. Models made to detect obfuscation rely on features left in the code by the transformation to make accurate decisions. Principles used in Adversarial ML attacks will allow us to identify the features being used to make decisions without having to analyze the models themselves. This information can then be used make changes to obfuscated programs to produce Adversarially Obfuscated samples that are more robust against detection. We also examine the concept of code expansion as separate defense

against detection. Code expansion would rely on adversarial ML to identify relevant features but would then add additional features to the sample to make features less unique without modifying the obfuscated code.

1.1 Research Questions

Recent research exploring the concept of using machine learning to assist in bypassing the protection of malware and other similar software has inadvertently created a potential security risk for legitimate software. This has prompted the hypothesis to explore if tactics used to undermine the effectiveness of machine learning may be used to enhance legitimate software security. This research seeks to answer the following research questions:

1. How reliably can obfuscation be detected using machine learning?
2. How easily can we evade ML detection using adversarial ML?
3. What are the constraints and possibilities for incorporating adversarial ML into an existing obfuscator?
4. Can code expansion achieve similar or comparable protection to adversarial ML?

1.2 Research Goals and Contributions

This research aims to provide effective methods to improve the stealth of obfuscation against adversaries employing machine learning to identify protections in place within software. The goal is to employ adversarial machine learning tactics alongside our own code expansion approach to arrive at satisfactory evasion rates.

We build upon previous work that has shown the potential for machine learning to

be used to accurate identify the type of obfuscating transformation applied to a piece of software without need for source code [14, 17, 18]. The methods we are implementing seek to enable the protections of legitimate software to not be weakened be adversaries who would benefit from this prior research.

At completion of this research, we hope to produce a prototype of our own obfuscator that can automate the processes we develop by incorporating them at the time of obfuscation. This tool will be made available to other researchers to serve as a basis for new research that tests it in order to enhance it in meaningful ways.

1.3 Document Outline

The remainder of this prospectus is outlined as follows: Chapter II discusses background concepts relevant to the work being done such as obfuscation, machine learning, and adversarial machine learning. Chapter III is an examination of the related work that is adjacent and foundational to our research. Chapter IV presents a timeline and the activities involved in the dissertation research.

CHAPTER II

BACKGROUND

This chapter will discuss several concepts in depth that are needed to understand the goals of this research. The section will begin by discussing the software protection approach known as obfuscation with a focus on the different types and the metrics used to evaluate them. Then, an overview of the basics of machine learning and several different types of models will be provided. Finally, the section will conclude with an overview of adversarial machine learning.

2.1 Obfuscation

The process of transforming the source code of software to make to it incredibly difficult for attackers to successfully analyze the code is known as obfuscation [5, 6, 7]. It is not wholly uncommon for obfuscation to be mistaken for "security by obscurity" in some areas [5]. Obfuscation is a part of a category of techniques that are to defend software against analysis and reverse engineering, particularly in Man at the End (MATE) attack scenarios [5]. In these attacks the malicious user has complete access to the software and control of the execution environment [6, 7]. As previously noted, these techniques are not complete protection, but instead focus on making it extremely difficult for the malicious user to successfully perform their attacks successfully [20]. Obfuscation

is accomplished through various transformations applied to the source code. This section will provide background on the concept of obfuscation as well as descriptions for the transformations used in this research. The metrics used to evaluate transformations will also be discussed as well as a look at the landscape of obfuscation focused research.

Obfuscation is accomplished by using a transformation, T, on a program in some representation (binary, intermediate, high level, etc.). T then produces a new program, P', which is semantically equivalent to the original program, meaning that for a transformation to be valid T(P) = P'. P' is most often outputted in the same form as the original program. In order to achieve a satisfactory level of protection, multiple different kinds of transformations, even multiple iterations of the same transformation, will usually be applied to the original program. These transformations will be both layered on top of each other and applied to different parts of the program. Obfuscation is rarely performed by hand on large sources and is instead accomplished using programs known as obfuscators, which apply transformations based on the algorithms designed into them [1].

From this, we can define obfuscators, O, as programs that apply transformations, T, from a selected set, t, to a source program, P. For example, given t = {T1, T2, T3, T4, T5}, an obfuscator O acts as a transformer (where O(P) = P') through application of some number and order of transformations. Multiple variants of a program can be generated from the same set of transformations be reordering the sequence that the transformations are in applied in. For example, O(P) = T3(T1(T4(T5(P)))) = P' would produce a program that is syntactically different from O(P) = T1(T5(T4(T3(P)))) = P'. Adding to this the fact that every transformation can be modified using various inputs and settings before they

are applied, an obfuscator can ensure that is variant generated is unique enough from each other [21].

Researchers created a formal definition of an obfuscator in order to determine if a program could be obfuscated to create a program that gave no information about itself other than the input/output relation, a virtual-black box [22]. The work determined that this was an impossibility for general programs and suggested instead that computational indistinguishability should be the goal for obfuscation instead. This would mean that if given two programs that are different but functionally equivalent, their obfuscated variants should be indistinguishable from one another [22].

The goal of obfuscation is to alter a source program in a way that creates a new program that is functionally identical to the original but is now more difficult to understand. The semantic equivalence between the original and the variant can be stated as: □x: $P(x) = P'(x)$. This all done to defeat man-at-the-end attacks performed by malicious end users who have unrestricted access to deployed code. Several commercial and open-source obfuscators are available and are in active use by software companies and researchers across the board [23, 24, 25].

An important component of obfuscation that must be remembered is that, while the obfuscating transformations themselves are not a secret, when where and how they are applied in the program are kept hidden. This further increases the effort and computing resources required since the adversary must first find the transformation and the determine which one it is.

2.1.1 Obfuscation Transformations

Obfuscation transformations are traditionally divided into three categories based on the workings of the transformations [1]. The categories are layout transformations, which focus on making source-code unreadable; data transformations, which focus on replacing data structures; and control transformations, which manipulate control structures. It is possible for transformations to be classified as dynamic transformations, which means that the transformation will be applied to the program at runtime [1]. Outside of this property, dynamic transformations can be placed into one of three preceding categories.

The following sections will give overviews of the transformation categories, as well as the transformations within those categories that are relevant to this research. Those transformations are Virtualization, Just-in-time Compilation (Jitting), Opaque Predicates, Control-flow Flattening, Encoding Literals and Encoding Arithmetic. As our work is not concerned with layout or dynamic transformations, these sections will be brief and are here for completeness' sake.

2.1.1.1 Layout Transformations, also called lexical transformations, differ from the two remaining categories by being concerned only with the readability and layout of source code [6]. In other words, they only alter the appearance of programs they are used on without any impact on semantics. The nature of these mean that they are one-way transformations. Once applied, the original form cannot be recovered. Examples of these transformations include removing comments, the reformatting of source code, and the changing of variable names.

Since the semantics of the program has not been altered, layout transformations have almost no cost impact and the general unintelligibility of transformed code means that the potency of these transformations is high as well; however, resilience and stealth are poor with many of the transformations [14]. The major drawback of these types however is that the transformations effect only source code and are erased with compilation. Nevertheless, layout transformations remain popular with JavaScript and other noncompiled languages.

2.1.1.2 Data Transformations are concerned with altering the data structures within a program to hide both what they are used for and their operations [26]. Collberg et al further subdivided these transformations by describing them as performing storage, encoding, aggregation, or data ordering [6]. A transformation as described as storage if it alters the container that holds the data in question. On the other hand, encoding transformations operate by changing what the data appears as, often by changing data types or creating functions specifically to store and return the data [14]. A transformation is an aggregation if it combines variables or structures into larger or more complex forms, thereby preventing an adversary from being able to immediately discern their usage [14]. Lastly, ordering means that a transformation alters the order of parameters or functions inside of classes and variables within method headers.

The only data transformation of interest to us within this research is of the encoding type. Specifically, we are interested in the encoding of string and integer literals. For strings, encoding is often accomplished by getting the value via function instead of storing it while integers use opaque expressions for encoding [27].

2.1.1.3 Control Transformations are the final category. As stated previously, this category contains transformations that alter the control structure of programs [6]. Like the data group, this category can be subdivided into three groups based on the effect had on the code. A transformation in this group with the effect of aggregation would either break down control structures, computation, or methods into multiple smaller units, or it could have the opposite effect and combine unrelated pieces of code into one [6]. Transformations of this type can also replace method calls with the full body of the method or vice-versa [7]. Figures 2 and 3 show examples of aggregating transformations.

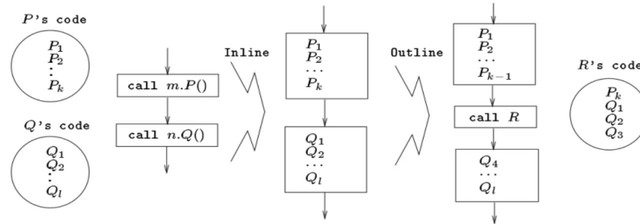

Figure 2. Inlining and Outlining of transformations [6].

```
int modexp(int y,int x[],
           int w,int n) {
   int R, L;
   int k = 0;
   int s = 1;
   while (k < w) {
      if (x[k] == 1)
         R = (s*y) % n;
      else
         R = s;
      s = R*R % n;
      L = R;
      k++;
   }
   return L;
}
```

```
void f(int xk,int s,int y,
       int n,int* R) {
   if (xk == 1)
      *R = (s*y) % n;
   else
      *R = s;
}
int modexp(int y,int x[],
           int w,int n) {
   int R, L;
   int k = 0;
   int s = 1;
   while (k < w) {
      f(x[k],s,y,n,&R);
      s = R*R % n;
      L = R;
      k++;
   }
   return L;
}
```

Figure 3. Splitting one function into two [5].

An ordering transformation would alter, potentially randomly, the order in which parts of the code are executed [6]. For example, statements could execute earlier or later than intended, or entire loops may be completely reordered. Since transformations of this type can heavily impact the semantics of a program, they must be heavily checked and validated to ensure the program still functions as intended [28]. To mitigate this risk, transformations of this type are usually performed on independent blocks of code [14]. The final type in this category is computational transformations, which operate by inserting useless code into the program [6]. Changes introduced by these transformations obscure the proper control flow as the inserted code will appear as possible alternate paths when, they have no purpose. This can be used to make the possible paths through the program seemingly grow at an extremely high rate [6]. The bulk of transformations we are concerned with in this research comes from the control category. These transformations are Virtualization, Just-in-time Compilation (Jitting), Opaque Predicates, Control-flow Flattening, and Encoding Arithmetic.

2.1.1.3.1 Virtualization is a type of ordering transformation that operates by creating a unique virtual environment within the transformed software [29]. This is accomplished by constructing an interpreter, usually in the form of a large switch statement, that is complete with its own unique instruction set. After the interpreter is created and placed within the software, target parts of the binary will then be transformed from their original source into commands from the interpreters newly generated instruction set [5].

Virtualization has shown to be quite effective against reverse engineering attacks making it a popular option for software protection [30-33].

<u>2.1.1.3.2 Just in Time Compilation,</u> called Jitting for short, is another transformation of interest to us in the control category. As the name implies, this transformation effectively adds dynamic compilation to the transformed program. When the transformation is applied, target statements will be replaced by method calls unique to that statement [7]. Unlike similar aggregate transformations, these functions do not simply contain the original code, but instead contain code that will compile and load into memory the original statement at runtime [34].

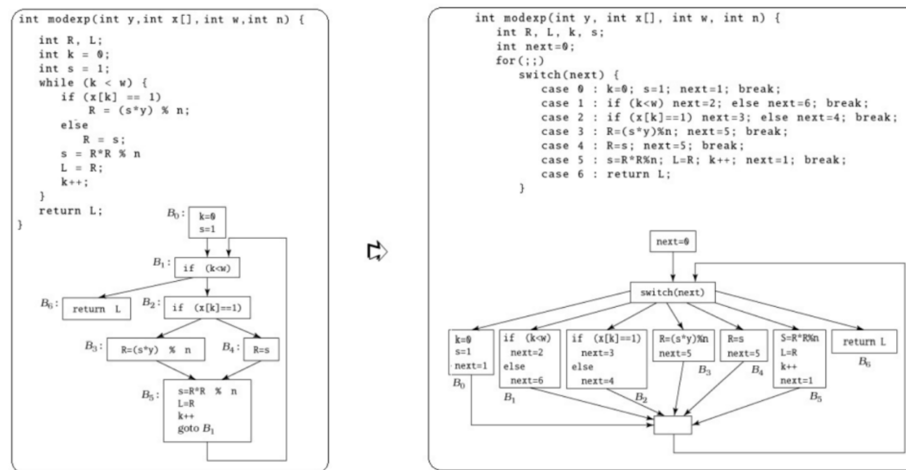

Figure 4. Control Flow Flattening [5].

<u>2.1.1.3.3 Control Flow Flattening</u> is a transformation that alters the flow of the program by combining areas of code containing branch statements into one large seemingly infinite loop [7]. This loop will be controlled by one segment of code, the dispatcher, that will determine the next code block to be executed. This allows the program to still be executed in the proper order despite the now flattened control flow. Once the code has been executed properly, the dispatcher will terminate the loop on the

next cycle, allowing the control flow to continue as normal [6]. Figure 4 shows an example of source code before and after it has been flattened.

2.1.1.3.4 Opaque Predicates is a transformation that can be classified within the control group as computational. When applied to code, multiple new branches will be introduced at points, giving the appearance that the code now has multiple execution paths in addition to the original [7]. The code comprising these branches is dummy code that performs no action relevant to the program and will never be executed. The new branch statements introduced for the paths will contain opaque predicates, predicates that will always evaluate to the same value, which are crafted by the obfuscator to appear as though they could be true or false [6, 35]. These dead branches can then not be immediately ignored by reverse engineering tools and make processes such as disassembly more difficult [6]. Figure 5 is an example.

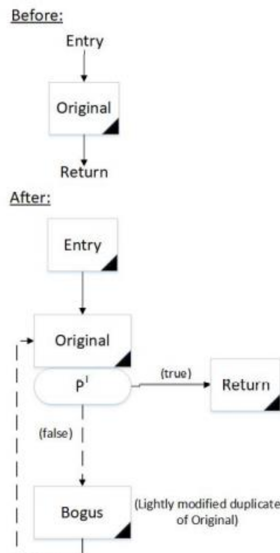

Figure 5. Opaque predicate used to introduce bogus control flow [17].

<u>2.1.1.3.5 Encoding Arithmetic</u> is a transformation which replaces mathematical expression found in the source with longer, more complex calculations [6]. This a type of aggregate transformation within this category.

2.1.2 Metrics for Obfuscation

In addition to semantic equivalence, a transformed program *P'* must be satisfactory in four other areas to be considered fully obfuscated [1]:

- Analysis and modification of *P'* should require more time than the original
- Construction of automated tool to analyze *P'* should be more difficult than the original
- Increases to time and overhead should be minimal
- *P'* should have the same statistical properties as the original [36].

These requirements are the basis for the metrics that are used to measure and compare obfuscating transformations: potency; resilience; cost; and stealth [5, 6, 7]. The requirements can be reworded and expanded to serve as informal definitions for the metrics. Going further, three of the above metrics can be given formal definitions.

The first metric, potency, measures how difficult a transformation makes it to understand obfuscated code as compared to the original. Put another way, potency is a transformations impact on the complexity of the source code [5, 6, 7]. Complexity in this scenario can be a measure of some aspect of the code or the performance of an analysis done against the code. As this means complexity can be measured in multiple ways, it becomes important to view the potency of a transformation with respect to the effectiveness of a transformation [36]. We can describe effectiveness by modifying the description of potency so that the effectiveness of a transformation is a measure of its

impact on a given complexity metric. Potency can then be defined using effectiveness such that the potency of a transformation is a measure of its effectiveness across a set of complexity metrics.

To formally define effectiveness, let *T* be a transformation, *P* be a source program, *P'* be an obfuscated variant of *P* such that *T(P)=P`*, and *C* be a given complexity metric for a program. The effectiveness of *T*, $T_{eff}$, can be obtained by taking the difference between *C(P`)*, the complexity metric of *P'*, and *C(P)*, the complexity metric of *P*. Viewing this as a formula, $T_{eff} = C(P`) - C(P)$. Using this formula a transformation can be labeled: effective if $T_{eff} > 0$, meaning it increased complexity; ineffective if $T_{eff} = 0$, meaning there was no meaningful change; and defective if $T_{eff} < 0$, meaning it decreased complexity [5, 6, 7].

Based on the definition on the formal definition for effectiveness, a transformation *T* can be considered potent against a set of complexity metrics *C* if for an obfuscated program *P'*=*T*(*P*), there is one or more $C_i \in C$ such that *T* is effective with respect to *P'* and the measurement of C_i, while for all other $C_j \in C$ (not equal to C_i), *T* is not defective with respect to *P'* and measurement using C_j. The potency of *T*, $T_{potency}$, can be viewed as the collective increase across all metrics in *C* [5, 6, 7]. In other words, a transformation is potent if, with regards to complexity, it increases one or more metrics while not decreasing any other metric, and a transformations potency can be measured as the increase in all complexity metrics.

The second metric, resilience, can be informally defined as the strength of a transformation against deobfuscation, via the use of an automated deobfuscator [5]. In other words, it is a measure of how time and resource intensive it would be for an

automated tool to deobfuscate a transformed program, undoing the transformation and producing something equivalent to the original program. In addition, resilience also considers the time and resources needed for an actor to create the tool. These two values are called deobfuscator effort and programmer effort respectively, with effort being a combination of time and resources [5].

With this description in mind, for a given transformation T and a program P, the resilience of T when applied to P is found by the formula:

$$T_{res}(P) = Resilience\ (T_{Deobfuscator_Effort},\ T_{Programmer_Effort})$$

Resilience is a qualitative measure that uses a scale of trivial, weak, strong, full, and one-way [5]. It should be remembered that all obfuscating transformations can be undone with enough effort, meaning that even one-way transformations simply require an extremely high amount of effort [20].

The third metric is cost, and it can be viewed as the simplest to understand and define. Cost is defined as the overhead introduced by the transformation after it is applied to the program [5]. It is a measurement of the added time needed to execute the program and the space needed for the program. Transformation cost (T_{cost}) w.r.t program P was formally defined as follows [36]:

$$T_{cost}(P) = \begin{cases} \textit{dear} \text{ if time/resources takes to execute } P' \text{ is } \textit{exponential} \\ \textit{costly} \text{ if time/resources taken to execute } P' \text{ is } O(n^p) \\ \textit{cheap} \text{ if time/resources taken to execute } P' \text{ is } O(n) \\ \textit{Free} \text{ if time/resources taken to execute } P' \text{ is } O(1) \end{cases}$$

The final metric is stealth. Stealth is defined as how well the transformed parts of the code are concealed in the original code. In other words, how closely do the obfuscations resemble the non-transformed areas of code [5]. Stealth is the only metric without a true formal definition. This is due to the nature of stealth being hard to measure and largely context sensitive.

2.2 Machine Learning

Machine Learning (ML) is an area of computer science that focuses primarily on the automation of analysis for data sets of varying sizes [37]. Through this analysis, models can be produced that reflect various relationships found in the data and can be used to make decisions for new data [37]. This section is not intended to serve as a complete overview of machine learning, but rather to provide a basic background and reference. Terms explained in this section will be used throughout the remainder of this dissertation. The following subsections will introduce the categories of machine learning, the basics of data collection, and provide overview for the training and testing process.

2.2.1 Categories

ML can be divided into 3 broad and distinct learning categories: supervised, unsupervised, and reinforcement. The main differentiation between the three categories are the types of data that is needed/being used for analysis and the expected output [38]. Beyond that each category also has techniques and algorithms unique to that class and well as different types of models produced at the end; though it is possible for models produced by the different categories to be used for similar applications. We will briefly

describe each of the categories, though for this research we are only concerned with supervised learning.

2.2.1.1 Supervised Learning is used for the analysis of labeled datasets. Samples in these datasets all have labels attached to them which represent the output that should be obtained as a result of analyzing that sample [39]. The goal of supervised learning then is to analyze the given samples of each label then learn how the features of those samples resulted in them being given that label. The result of this is a model that can take in samples and then properly label them, even if the samples were not part of the original dataset. This process can be given one of two names depending on the nature of the label. If the label places the sample into a category, group, or class it's called classification. Examples of classification are malware detection, translation, and spam filtering [40, 41, 42]. If the label represents a continuous numerical value, then it's called regression. Examples of regression include predicting an employee's potential salary and predicting the expected value of house [43, 44].

2.2.1.2 Unsupervised Learning, contrasting the previous category, is used for the analysis of datasets comprised of samples with no labels. The common goal of this kind of learning is to use a chosen similarity metric to find the distance between samples in the set and then use this distance to create groups that can then be labeled [45]. This kind of task is known as clustering. It is possible to use unsupervised learning to assist in the training of other models [46].

2.2.1.3 Reinforcement Learning, is used to produce an agent that can act autonomously within a given environment or to serve as a policy for creating planning and control schemes [47]. To that end, datasets for this task take the form of "runs" through a

simulation of the environment in question. Each sample in the set will take the form of the sequence of actions taken in the run and the rewards that were received. Unlike the previous methods, this kind of dataset does not need to exist prior to learning beginning. The agent can be placed into the simulated environment and learn from scratch as it tries new actions [48].

2.2.2 Phases

Machine Learning of any category can be broken down into three phases. These are data collection, training, and testing. The following subsections will walk through these phases and give a basic overview of each. As mentioned earlier, our research is

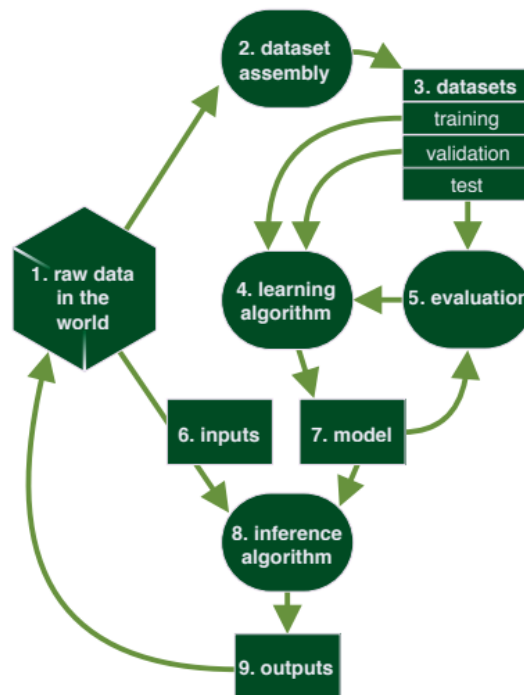

Figure 6. Diagram of a generic ML system. Arrows represent information flow [49].

focused only on supervised learning so these sections will focus examples primarily on that category. Figure 6 shows a general view of a ML system.

2.2.2.1 Data Collection primarily consists of gathering a large number of samples that will be analyzed to form a dataset. The type and nature of this data will be dependent on the problem that is being solved and the learning style being used [38]. For an application of supervised learning, such as spam filtering, the dataset would consist of emails that would be labeled as either "spam" or "not spam" [42]. A dataset for unsupervised learning, such as one being used for anomaly detection, would consist of samples that described events with no label attached to them [50]. The goal here would be to use ML to find the events in the set that are outliers or anomalous. Finally, a dataset for reinforcement learning that would be used to train an agent to play a video would be playthroughs of the game made up the possible states of the game, what actions were taken in those states, and the result of those action [47].

2.2.2.2 Training and Testing are the next two phases. The training phase is where the data is analyzed to learn the model [51]. Generally, a ML model can be described as a parametrized function that takes in a sample as input. The sample can be given to the model in its original form, or it can represent as a set of features that describe the original data. The output of this function is then the predicted answer for whatever question is being asked or the value for a property of interest. The goal of the training phase is for the learning algorithm that was chosen to find the correct parameters for the function [51]. The parameters are determined based on the category of learning being used. For supervised learning the parameters are modified so that the predictions of the model

match the labels from the dataset. This is accomplished through a loss function which gives a value for how dissimilar the models prediction is from the correct label [51].

After training is complete, the next step is testing. In supervised learning, this is done by using the created model to predict labels from a test dataset and scoring its performance. This test dataset is comprised of samples that were not included in the training data. This is done to validate that the model can perform sufficiently on data that was previously unseen.

2.3 Adversarial Machine Learning

As ML became more and more prevalent, it became clear that the reliance on large amounts data at the training phases and the uncertain nature of new data at the testing/inference phases presented unique security challenges [52]. Chief among these was the potential for an adversary to manipulate the data to achieve a variety of ends such as impacting performance, or the stealing of sensitive data previously seen by the model. The study of theses security issues gave rise to the field of Adversarial Machine Learning (AML). AML research focuses on the attacks used on ML systems and the capabilities required for them; improving the design of and adding defenses in order the mitigate those attacks; and the overall consequences of a successful attack [19, 52, 53]. AML is not concerned with the study of flaws and biases that may impact a system as they are not intentional attacks [49, 19].

In this section we will first focus on describing the taxonomy of an AML attack by examining the ML attack surface, the techniques available in AML, and the different knowledge levels an attacker can have of a system. We will then provide more detail on

Evasion Attacks, the type of attacks used in our work, as well as detail on the crafting of adversarial examples. As defense is not relevant to our work, we will not be discussing the various defenses used in AML.

2.3.1 Machine Learning Attack Surface

An attack surface is the different locations in a system that can be targeted by an attacker. One goal of security is to try and keep the number of targets in systems attack surface as low as possible as these targets can allow an attacker to manipulate input, impact system functionality, or steal data [49]. While the attack surface of a specific ML system will vary slightly based on the type of learning being used, purpose, and implementation of the system, it is possible to model an attack surface for AML based on the general design of an ML system and the ML pipeline [39, 19]. The NIST taxonomy for AML attacks defines the attack surface (called 'target' in the taxonomy) using the ML pipeline and identifies three target areas: physical domain, digital representation, and the ML model [19]. Figure 7 shows the ML pipeline with these targets labeled.

The first attack target is the input portion of the physical domain. This would be the inputs collected from sensors, users, or a data system. Following this, the next target area is the digital representation. This is the portion of the pipeline where inputs that have been collected are converted into the proper data form and then preprocessed before being given to the ML model. The next target is the ML model in use and encompasses the model being given the features taken from the input and producing an output. The end of the pipeline is part of the physical domain attack surface just like the beginning but deals with the handling of output by the system instead of inputs [19]. There is one more

attack target not shown in the pipeline model and that is the data collection process as well as the training stage of the model [19, 54].

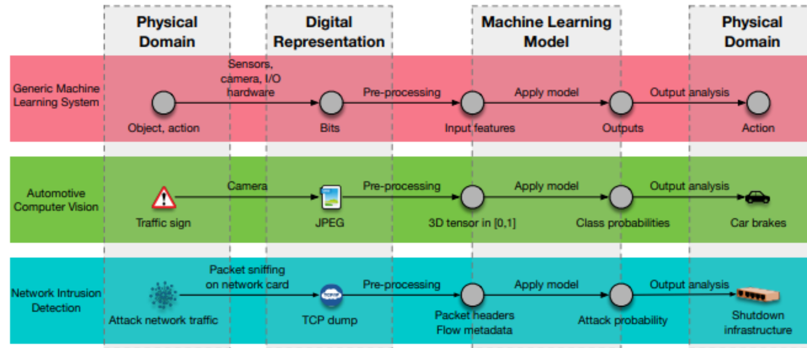

Figure 7. ML pipeline with labels attack surfaces. The first row is generic while the 2nd and 3rd rows are for specific systems [39].

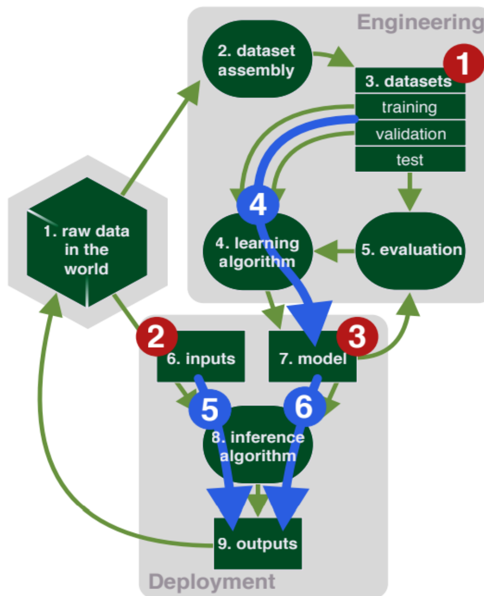

Figure 8. Known attacks and attack surfaces on ML systems [49].

A different description of the ML attack surface is given in [49] based on a more detailed but still generic overview of an ML system can be seen in Figure 8. Manipulation attacks are pictured in red at the site of attack: (1) data manipulation. (2) input manipulation. (3) model manipulation. Extraction attacks are pictured in blue, showing the flow of information: (4) data extraction. (5) input extraction. (6) model extraction. Attack surfaces roughly correspond to gray plates: deployment, engineering, and data sources [49]. While this model only has three target areas and was not created specifically for AML, it does not contradict the NIST taxonomy and can be seen as complimentary.

2.3.2 Techniques

After identifying the target area of the ML system, the next component of an AML attack is the technique being used for the attack. The technique used in an attack will be chosen based on multiple factors, namely the chosen target and goals of the attacker. Techniques are generally classified first into two groups based on the target and then into smaller groups based on the goal of the attack [49, 19]. The following subsections will describe these two broad categories as well as the subcategories within each. For this dissertation we will use the six categories from [49] and as shown on Figure 8. Again, while not intended for AML, the subcategories generally used and as shown on the NIST taxonomy fit well into these categories. We will briefly describe categories and techniques but will not go into detail. Readers wishing to know more about specific techniques are encouraged to view the NIST AML taxonomy [19] which contains references for all recognized techniques.

<u>2.3.2.1 Training</u> category techniques are used for attacks that target the data collection or training phases of an ML system. All techniques in this category fall into the first category of Figure 8. Also called “causative” and “poisoning” techniques, the goal of attacks using these techniques is to corrupt or influence the model so that once trained it behaves differently than what would be expected [55 ,56]. To accomplish this, techniques operate on either the training data being used or tamper with the settings used for the models training. The latter techniques are called *Logic Corruptions* and can modify the ML algorithm in order to alter the entire learning process [57, 58].

Attacking the training data is accomplished through either *Data Injection* or *Data Manipulation* [19]. Injection techniques insert new data into the training set that has been modified in some way so that it will cause the model to learn incorrect input output mapping [59]. Manipulation techniques modify data that is already present in the training set with the same end goals. The data introduced or modified as part of these two techniques can accomplish the goals of the attack even if the data is simply noisy and not malicious [60]. Models that continue to train after deployment, known as online models, can be targeted by these techniques at initial training or after deployment [19, 54]. The unique training nature of reinforcement learning has been shown to be vulnerable to these types of attacks as well [61].

<u>2.3.2.2 Inference</u> techniques are all other techniques that target the model after it is completed and deployed and are labeled inference techniques. Attacks using these techniques are known as *Exploratory Attacks*, and do not tamper with or change the target model and focus only on getting a chosen output or learning about inputs or

training data [55]. Attacks using these techniques are the most often researched aspect of AML [49, 19].

The goal of changing a model's output is accomplished via *Evasion*. Techniques used for this purpose craft what is known as "adversarial samples" by modifying existing samples using calculated noise in order to shift that sample across a model's decision boundary [62, 63]. Figure 8's input manipulation category is the equivalent to evasion techniques. Techniques used for *Model Inversion*, use the given output responses of a model to learn information. This can be unseen inputs, data from the training set, or if a given sample was part of training set [64, 65]. These techniques are part of groups 4 and 5 on Figure 8 depending on the type of information being gained.

The remaining techniques in this category are used for *Model Extraction* and belong to group 6 on Figure 8. These techniques query the system under attack in order to reconstruct the ML model using the information gained from the input-output pairs [66, 67]. Any technique in the Inference category can also be described *Oracle Technique* if the only interaction with the ML system is the querying of chosen inputs and then analysis of the outputs [19, 68].

2.3.3 Adversary Knowledge

The last component for defining an AML attack is the knowledge possessed of the ML system by the adversary [49, 19]. In this instance, the adversary's knowledge also encompasses the level of access they have to the system in question [63]. Much like attacks in other security or analysis areas, attacks in AML are placed on a scale where one end is an adversary having complete knowledge of an ML system, *Whitebox Attacks*,

and the other end is an adversary with only the bare minimum of knowledge, *Blackbox Attacks* [69].

If an attack falls into the Whitebox scenario, an adversary can be assumed to know any relevant information about the system to perform the attack to its full effect [49, 19]. This can include knowing the nature of the training data used, specifics of how the model was trained, the parameters found and used for the model's function, the nature and features of input data as well as output data, etc [19, 63, 69]. When viewing Whitebox attacks from an access perspective, the adversary is assumed to have access to any relevant step of the ML system [57]. It is possible for an attacker to obtain Whitebox access to a deployed model through a variety of means such as reverse engineering the deployed system [70].

Blackbox scenarios on the other hand, involve attackers whose knowledge of the system includes only the information which can be obtained from using the ML system [58]. More specifically, Blackbox attacks are assumed at most to only have knowledge of the output for certain inputs; however, the attacker's knowledge of the inputs and outputs is often still limited to the raw data going into the system and the systems provided output. They may still lack knowledge of any changes performed on the raw data, and the knowledge of the output is not assumed to include the actual output of the model, just the system [63, 57, 58].

In practice attacks often fall into a middle ground called *Graybox Attacks*, in which the attacker has some combination of knowledge relevant to the attack [19]. Figure 9 provides an example of the aspects the adversary may need knowledge of for training and inference techniques.

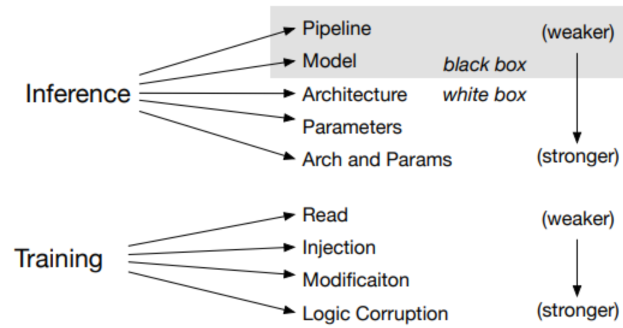

Figure 9. This figure shows the components of an ML system. Parts that can be attacked by the two categories as well as what knowledge is generally required [39].

2.3.4 Evasion Attacks

For the work done in this dissertation, we are concerned with the category of attacks with the goal of evading the model. These *Evasion Attacks*, as mentioned previously, are performed by feeding selectively modified inputs known as Adversarial Examples (AE) that will cause the output of the model to be different than it would have been for the original input [62, 71]. The changes made to these samples are known as adversarial perturbations and act as noise to confuse the ML model by altering the features the model will process [63]. While evasion attacks can be performed against unsupervised and reinforcement learning, the most often studied attacks are shown against supervised learning models, particularly those performing classification [49, 19, 72]. As our work is only concerned with classification, this section will only discuss evasion attacks of that nature as well.

Figure 10 gives a visual example of various AE created for evasion attacks for different kinds of inputs. The noise added to samples is determined the input-output relationship of the model being targeted [55]. Finding the noise needed to create an

adversarial example has been formalized as the minimization problem:

$$\underset{r}{arg\ min}\, f(x + r) = l \ s.t. \ \ x^* = x + r \in D$$

where x, an input correctly classified by f, is modified with some perturbations r, to produce an adversarial sample x^* that is part of the same input domain D but will now appear as a new label l [73]. The attacker can either choose l or let it be any label other than the original, resulting in the attack being targeted or nontargeted respectfully [74]. The type of analysis that can be performed is dependent on whether the attacker has Whitebox or Blackbox knowledge of the model [75].

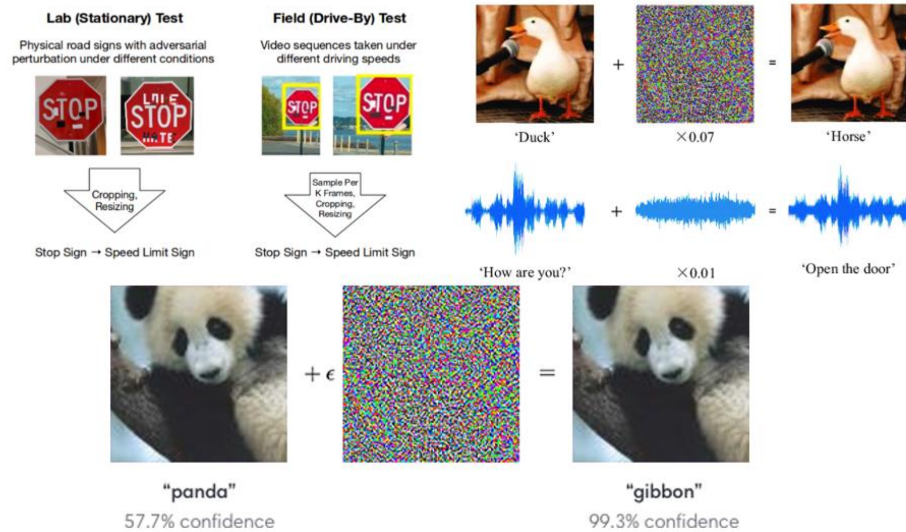

Figure 10. Four examples of adversarial inputs.

2.3.4.1 Whitebox Evasion consists of attackers performing evasion attacks with Whitebox knowledge can analyze varying aspects of the function *f* that represents that ML model as well the parameters θ for that function [69]. This knowledge allows different methods to be used to solve the problem for creating an AE, with this being the primary difference between Whitebox attacks. The first techniques shown made use of optimizers already in use in machine learning as the approximation to solve the problem [76, 71]. Szegedy et al. used the L-BFGS optimizer and were the first to find that ML models, including deep neural networks, were misled by *r* values that were not easily seen by human observers [77].

The remaining techniques make various assumptions to craft samples more efficiently. The first of these techniques was the Fast Gradient Sign Method [78]. This technique makes an assumption about the linearization of the model resulting in the equation for making an adversarial sample:

$$x^{*}=x+\epsilon \cdot sign(\nabla x J(\theta,x,y))$$

where *J* is the cost function used for *f* [78]. This technique has proven to highly effective at crafting samples in a variety of instances [19]. Many other techniques that make other assumptions or constraints have been created as well [63, 57, 71]. For instance, if there is a need to place limits on the changes introduced via *r*, there are techniques for varying levels of constraint [74, 79, 80]. The bounds placed on the changes allowed for a sample are often determined based on the domain [70].

In this research, we intend to utilize the Fast Gradient Sign Method (FGSM) and the Carlini & Wagner Method (C&W) to generate adversarial samples [81]. We will describe both methods in a later section.

2.3.4.2 Blackbox Evasion. Many challenges arise when evasion attacks are attempted in a Blackbox setting. Without knowledge of internals such as the gradient for the target model, the methods used for crafting AE cannot be used [71]. The only guaranteed information in this attack scenario is the output responses given by the system, which makes these attacks similar to reconnaissance attacks used to probe a system for info [58]. With this in mind, AML borrows the concept of an oracle from cryptography and uses this to describe the most common type of Blackbox attacks. The oracle in these attacks is the target ML system. The attacker can query the oracle with any valid input and receive the output from the oracle but nothing else [58]. Even with this limitation, oracle attacks are still effective as a large amount of information can be gained from just observing the output [68].

Methods used to craft adversarial examples via an oracle, are compared by the amount information that gain be observed with respect to the number of queries. One such method uses the weighted difference between x to x^* associate a cost function with transforming x to x^* [82]. The problem then becomes finding the number of queries that results in the lowest cost for the modification. This was shown to work well for continuous values, such as those found in regression, but less so for discrete values, such as labels for classification [83]. If the output given by the system is the probability of a sample being a certain class, the number of details that can be recovered make crafting AE easier. With this kind of output the features of a given sample can be modified using genetic algorithms, where the fitness of samples is determined by the probability given by the oracle [84].

It was observed that if an AE created for a model was given as input to a different model performing a similar function, then that AE would often be misclassified by that model as well. This is known as the concept of adversarial transferability and has been observed even on models that were trained on different datasets [62]. This led to concept of the substitution attack model. In this model the attacker performs a model extraction attack by querying the oracle with chosen inputs to have the oracle provide the proper labels so that the attacker can construct a surrogate data set. The attacker then trains their own ML model on this data set in order to produce a model with a similar decision boundary [85]. Due to adversarial transferability, the attacker can then perform a Whitebox attack against this new model to create samples to be used against the Blackbox model. This attack has been shown to work with models trained on datasets not labeled by the oracle but that are similar in nature [86].

2.3.5 Crafting Adversarial Examples

As mentioned previously, we will be making use of the FGSM and C&W to craft adversarial samples for this research. In this section we will introduce both methods and explain their components and functions. Before this we will describe the machine learning algorithm known as a Deep Neural Network (DNN) as it will be the primary ML algorithm used in this work and describing it here will aid in the description of the two AE crafting methods.

2.3.5.1 Deep Neural Network. A DNN is a neural network that consists of an input layer, two or more hidden layers, and an output layer [37]. These layers are comprised of neurons which are connected to the neurons of the layer preceding and following, with the input layer having no preceding connection and the output layer having no following

connection [38, 51]. The connections between neurons have varying weights associated with them and all neurons after the input layer contain a chosen activation function [51]. When a sample is given to a DNN the different features of that sample are each given to a corresponding neuron in the input layer, then sent along that neuron connections to the neuron in the next layer. Values will be modified with the weight of the connections between the neurons and will be used as input in the new neuron's activation function [37, 38, 51]. The value of that will then be sent to the next layer until the final values are given from the output layer. Figure 11 shows an example of a DNN.

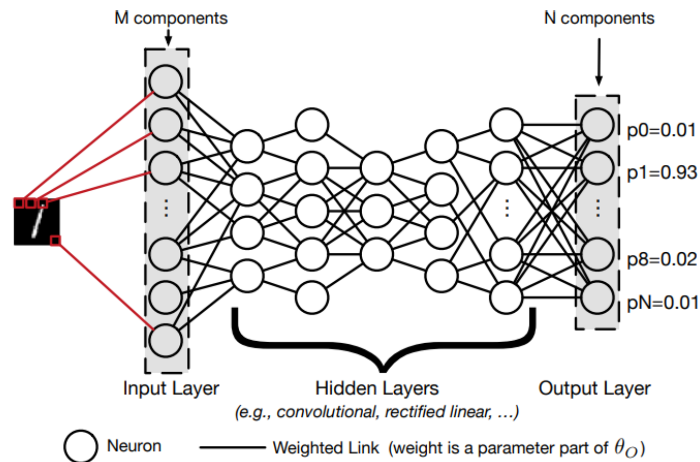

Figure 11. Example of a DNN. Output being the probability of the input being in one of N classes [87].

As a DNN is trained the weights of the connections are adjusted to accomplish the model's learning. For classification, the weights are adjusted by on a loss function that compares the expected output to the given output and then adjusts the weights through

backpropagation or another means. For classification task, the output layer will consist of an equal number of neurons to the possible classes. Output is given in the form a probability, with each neuron showing the probability that the input is of that neuron's respective output. This is sometimes called confidence.

To align with the view of an ML used previously, we can view a DNN as a composition of parameterized functions equal to the number of layers. The weights of a layer's connections form the parameters θ for that layer's function. This allows us to view our DNNs function as:

$$f(x) = fn\ (\theta\ n, fn1\ (\theta\ n1, \ldots f2\ (\theta\ 2, f1\ (\theta\ 1, x))))$$

This large number of parameters and the common failure of neural networks to properly generalize learning results in DNNs being vulnerable to most of the methods used to craft AE [62, 78, 74].

2.3.5.2 Fast Gradient Sign Method. The FSGM is a method used to solve the optimization problem of crafting AE using approximation [78]. FGSM creates AE using the following function:

$$x^{*}=x+\epsilon \cdot sign(\nabla xJ(\theta, x,y))$$

where ∇xJ is the gradient of the loss function of a target model f, θ is the vector of all parameters from f, and y is the correct label of input x. The sign of the gradient is then taken and will return +1 if an increase in a feature will increase loss (the model's error) or -1 if a decrease in a feature will increase loss. This is then multiplied by the input variation ϵ. This value controls the intensity of changes: higher values result in more drastic changes to the sample while lower values are more subtle changes.

2.3.5.3 Carlini and Wagner Method. The Carlini & Wagner Method is a way to generate targeted AE that optimizes for misclassification while at the same time minimizing the distance between the original sample and the generated AE [81]. This is accomplished by limiting the scope of possible changes to the sample using a chosen distance measure. The base formula for the is:

$$min\ D(x, x^*)\ s.t.\ C(x + \delta) = t,\ x + \delta \in X$$

where D is the chosen distance function, δ represents the noise added to the sample, C is the objective function of a classifier, and t is the target class. The last part is a constraint stating that the created AE cannot be changed so much that it is no longer a viable sample from the set X.

The formula is then expanded and modified in order make the optimization easier and to solve issues related to the use of gradient analysis and the constraints of the formula. This results in three distinct attack methods using either the L_0, L_2, or L_∞ distance measures. For this work we choose to use the L_2 distance. When using this distance, the formula for finding δ is:

$$\delta = 1/2\ (tanh(w) + 1) - x$$

where *tanh()* is the hyperbolic tangent function and w is a variable that is optimized by:

$$min\ ||1/2\ (tanh(w) + 1) - x||_2 + c \cdot f(1/2(tanh(w) + 1)$$

where c is a chosen constant greater than zero and f is the function:

$$f(x^*) = max(max\{Z(x^*)_i : i \neq t\} - Z(x^*)_t, -\kappa)$$

where κ is a chose value representing confidence and *Z()* is the models raw, unnormalized class probability predictions for a given sample [81].

CHAPTER III

RELATED WORK

Our research is based on foundational work done by several other researchers that showcased using machine learning to identify obfuscating transformations applied to programs. These papers described below.

3.1 Metadata Recovery from Obfuscated Programs Using Machine Learning

Salem and Banescu proposed and showcased the concept that a ML model could be trained to perform metadata recovery on an obfuscated program [14]. Metadata recovery is a process performed, generally manually, by reverse engineers that involves analyzing an obfuscated program to identify the transformation used for obfuscation. Salem and Banescu proposed that since transformations leave uniquely identifiable side effects in programs, it would be possible to use ML to detect these changes [14]. Figure 12 is taken from the paper and gives an example of this idea. To the best of our knowledge this was the first paper to showcase this idea.

To test their proposal, the authors used 2 datasets of C programs in a series of classification experiments [14]. The first was a set a of 40 programs that consisted of various implementations of mathematical functions, array operations, and other operations that made use of a wide range of functions in the C language [88]. The second

and larger dataset was a set of 1920 C programs generated using Tigress obfuscator [24]. The programs all shared a similar template but differed in the operations performed within the program. Both data sets were then obfuscated using Tigress to produce obfuscated variants. The second dataset was obfuscated using the default options for 5 transformations (1920*5=9605), while the second used a variety of options for the 5, resulting in 39 variants of each program with some having to be discarded due to errors (39*5-90=1470) [14]. All programs were then compiled using GCC to strip the symbols table and relocation information and then again using GCC with default settings. This results in two variants of each dataset for four datasets total.

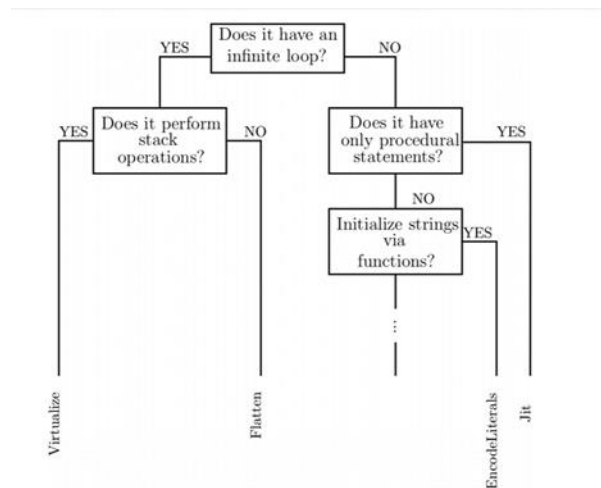

Figure 12. Flowchart showing side effects in code. These can be used to determine the transformation [14].

The features chosen for classification are the Term Frequency-Inverse Document Frequency (TF-IDF) of the opcodes within the programs. The authors extract opcode

both statically and dynamically giving each program two different feature sets. Two different ML algorithms were used to create models from the data: Naïve Bayes and Decision Trees [14]. The authors vary the hyperparameters for each, resulting in a total of four different models (two Naïve Bayes and 2 Decision Trees) for each feature set [14]. Figure 13 from the paper outlines this process.

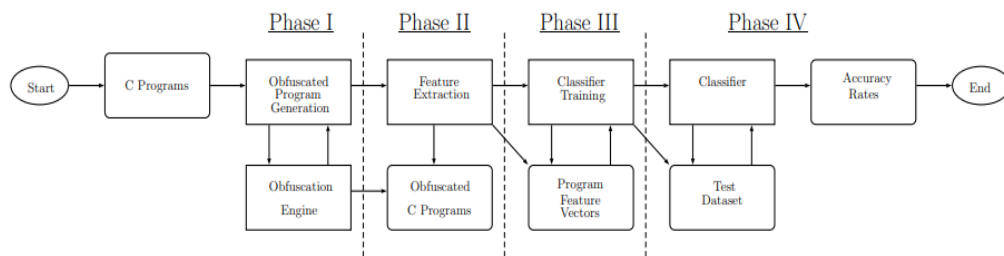

Figure 13. ML pipeline for [14].

Two experiments were performed in the paper. The first had the classifiers trained using 10-fold cross validation, while the second experiment used training and test sets with any variant of a program appearing only in one set [14]. This was performed with 10 different set variations with the accuracy being averaged. The results of these experiments for both datasets can be seen in Figures 14 and 15. Overall, the experiments showed that detection of obfuscation was feasible, provided that a program similar to the one being analyzed had been part of the training set.

Another contribution of this paper was the Oedipus framework. Written in Python, it is collection of scripts that makes use of various python packages and other software that can be used to recreate the experiments performed in the paper [14].

			Naive Bayes		Decision Tree	
			SelectKBest	PCA	Gini	Entropy
Static	raw	stripped	0.37 / 0.40	0.38 / 0.39	0.99 / 0.38	0.98 / 0.40
		non-stripped	0.38 / 0.40	0.96 / 0.40	0.96 / 0.25	0.96 / 0.38
	filtered	stripped	0.61 / 0.44	0.38 / 0.44	0.98 / 0.39	0.99 / 0.46
		non-stripped	0.86 / 0.40	0.40 / 0.40	0.99 / 0.44	0.99 / 0.61
Dynamic	raw	stripped	0.35 / 0.45	0.36 / 0.54	0.99 / 0.61	0.99 / 0.39
		non-stripped	0.60 / 0.40	0.35 / 0.42	0.99 / 0.56	0.99 / 0.40
	filtered	stripped	0.86 / 0.48	0.65 / 0.34	0.96 / 0.48	0.96 / 0.52
		non-stripped	0.92 / 0.55	0.62 / 0.35	0.99 / 0.57	0.99 / 0.41

Figure 14. Classification accuracies for experiments 1 and 2 (in red). Using 40 self-gathered C programs [14].

			Naive Bayes		Decision Tree	
			SelectKBest	PCA	Gini	Entropy
Static	raw	stripped	0.80 / 0.19	0.68 / 0.19	0.89 / 0.20	0.88 / 0.20
		non-stripped	0.80 / 0.19	0.69 / 0.19	0.89 / 0.20	0.88 / 0.19
	filtered	stripped	0.84 / 0.26	0.70 / 0.22	0.89 / 0.40	0.89 / 0.42
		non-stripped	0.87 / 0.23	0.78 / 0.20	0.89 / 0.35	0.88 / 0.34
Dynamic	raw	stripped	0.57 / 0.02	0.64 / 0.44	0.96 / 0.45	0.97 / 0.45
		non-stripped	0.57 / 0.02	0.63 / 0.44	0.96 / 0.45	0.96 / 0.45
	filtered	stripped	0.89 / 0.42	0.84 / 0.52	0.93 / 0.59	0.93 / 0.61
		non-stripped	0.96 / 0.36	0.91 / 0.14	1.00 / 0.61	1.00 / 0.61

Figure 15. Classification accuracies for experiments 1 and 2 (red). Using 1920 random programs [14].

3.2 ByteWise: A case study in neural network obfuscation identification

Jones et al. proposed and showcased that a neural network could identify and label the bogus control flow (BCF) of a program introduced as part of an Opaque Predicate [17]. Based on the idea that transformations only insert or delete code, they state that if a pattern exists across all of the inserted code in a program, then the inserted code can be identified at the byte level. This differs from the previous paper which focused on binary level analysis. The method and concept were based upon work done using neural networks to identify the boundaries of functions in a compiled program [17, 89].

To perform their approach, the authors create a variant of Obfuscator-LLVM (OLLVM) which they refer to as an annotating obfuscator. The purpose of this obfuscator

is to place unique markers within the basic blocks introduced as part of a bogus control flow to make identifying the blocks easier [17]. The annotated binaries are only used for the purpose of creating a labeled dataset. Every annotated binary has an accompanying unannotated binary that is identical.

The authors use programs written in C that were obtained from GitHub as the basis of their three datasets [17]. The three data sets are: Mono, containing 7 binaries obfuscated with a BCF; Duo, containing 14 binaries with 7 having a BCF and 7 not obfuscated; and Multi, containing 72 binaries with 7 having no obfuscation and the remaining having some combo of BCF, Instruction Substitution, and Control-Flow Flattening. For all three datasets the annotated versions are used to label the basic blocks of the unannotated versions as bogus or not. Once a binary has had all its basic blocks checked, a feature set is constructed with the values of the bytes from the basic blocks labeled either bogus or not. Each of the datasets has one large feature set instead of each sample having its own feature set.

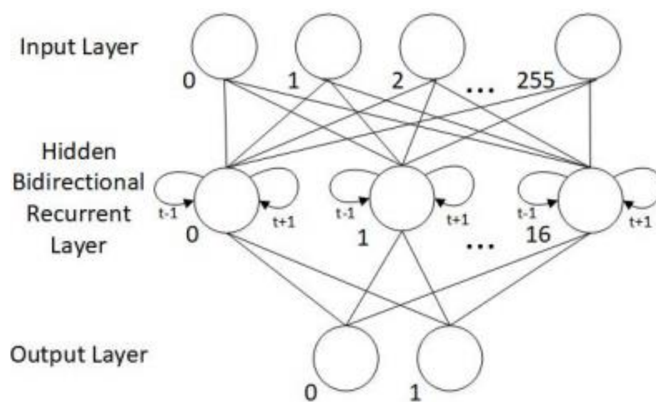

Figure 16. Structure of the RNN model used for the paper [17].

For the ML models, the authors train a recurrent neural network with a single bidirectional long-short term memory (LSTM) layer, and an input layer that is one-hot encoded [17]. This is shown in Figure 16 and this model is created for each of the 3 datasets.

Tests are then down for each of the dataset models and a final test is performed for an ensemble of the mono and multi dataset models. The ensemble test has the two models use confidence-based voting (the model with the higher confidence value in its prediction wins) to determine if a byte is bogus. Four test sets are generated using a different program from GitHub and used against each model and the ensemble. The results of these tests are shown in Figure 17. Across all four tests the voting model generally outperforms the other three, though often by a narrow margin [17]. These results showcase the authors claim that BCF detection with a neural network is feasible and could lead to automated code removal.

	Mono	Duo	Multi	Vote
Test Set 1 (F1)	0.92	0.98	0.98	0.98
Test Set 2 (Acc)	0.90	1.00	1.00	1.00
Test Set 3 (F1)	0.73	0.87	0.98	0.98
Test Set 4 (F1)	0.86	0.81	0.75	0.85
Avg	0.85	0.92	0.93	0.95
Std Dev	0.09	0.09	0.12	0.07

Figure 17. Results of BCF detection [17].

3.3 Fine-Grained Static Detection of Obfuscation Transforms Using Ensemble-Learning and Semantic Reasoning

Tofighi-Shirazi et al. expanded upon the work done in [14] to account for the layering of multiple obfuscations in a single program. In other words, they proposed and showcased a method to detect multiple transformations that are present within a single sample, as opposed to only detecting a single transformation [18]. They also showcase that their method can be used to detect variants of transformations, causing them to label their approach as fine-grained.

The dataset used for the experiments is identical to the dataset used in [14] and for each experiment the authors perform the same two types of cross validation from that paper as well [18]. The primary difference is in the features used for the samples and the models created from training. Figure 18 outline the process used in the paper from data extraction to training. The authors use disassembly to obtain the assembly instructions of an obfuscated function, then convert from assembly into the intermediate representation (IR) of the MIASM framework. Once converted, the IR instructions are then used to produce a symbolic execution trace, the output of which is then normalized to remove unique values and ID’s. This is done for every basic block in a function, with the output of each basic block being the raw data for a file.

To handle the multi-label detection problem, the authors examine two different ML methods [18]. The first is to use a single model to detect each type of obfuscation present. This second approach is to use a chain of classifiers where each is only trained to detect one type of classification and then gives its decision to the next classifier in the chain, with the final classifier making its decision based on the input and the output of

every preceding classifier. For both approaches, the authors also examine using an ensemble classifier instead of a single model. All models used are decision trees and random forest.

In the experiments performed, the models were able to achieve up to 91% accuracy for labeling transformations and up to 100% accuracy for labeling variants of a transformation [18]. Notably, the models performed much better in the experiments using the variant training than the models from [14]. The results are promising but still show room for improvement.

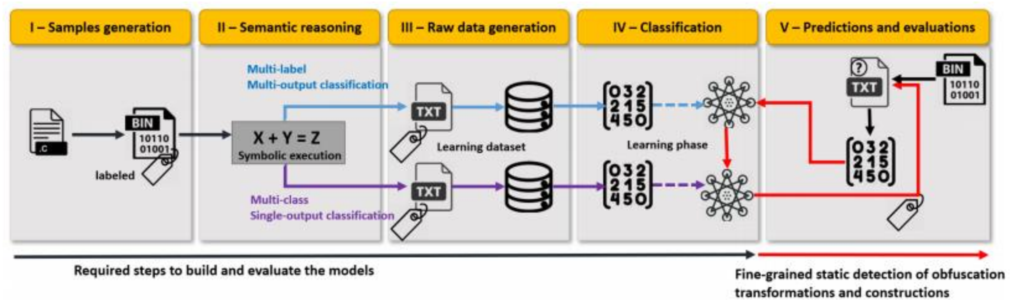

Figure 18. Design steps for detection system [18].

CHAPTER IV

METHODOLOGY

The main goal of this research is to propose and investigate two approaches which are called Adversarial Obfuscation and Obfuscation Expansion, which both make use of adversarial machine learning to some degree to evade automated detection. Both approaches will result in previously obfuscated programs that have been converted into adversarial samples to evade automated detection. Programs modified in these ways will have improved obfuscation stealth in the context of adversaries making use of machine learning. In a series of three phases, we will demonstrate the evasion efficiency of both methods while also comparing their impacts on obfuscation metrics other than stealth.

In our first phase, we implement a suite of machine learning classifiers which have been trained to detect and label obfuscation transformations. Each of these detectors will be trained and tested on the same data set but using different feature sets to provide diversity in the detectors. Next, we will begin using Adversarial ML to create sample programs that are able to evade detection by some or all the classifiers. Samples will be crafted based on both of our proposed methods. Generated samples will also be tested to see if the changes made had significant impacts on obfuscation metrics. Lastly, we will construct a tool will which automate the process of adding the changes of our proposed

methods at the time of obfuscation in order to maximize the costs and benefits. This will also be when we compare our proposed methods.

The work in this dissertation was done with the goal that each research question forms the basis of one or more research papers. With that in mind, that subsequent chapters for the three phases of our work are presented in paper format.

4.1 Dataset creation

For this research, we formed our dataset by first gathering a large quantity of source code for a variety of C programs. It was important that we obtain source code for these programs, as that would allow us to produce obfuscated variants. Our obfuscated variants were produced using the Tigress and OLLVM obfuscators.

4.1.1 Dataset source

The C programs used to create our obfuscated variants are the same as the ones used by Banescu et al in their obfuscation research, as well as other papers listed in our related work. [95]. The programs in this dataset consist of:

1. A set of 48 programs with few lines of code constructed by varying code characteristics such as: symbolic inputs, depth of control flow, number of loops, etc.
2. Programs automatically generated by the RandomFuns transformation of the Tigress C Diversifier/Obfuscator.
3. Non-cryptographic hash functions
4. Algorithms taught in Bachelor level computer science and programming courses.

This brings us to a data set of 5,136 source c files. Figure 19 gives an example of one of the programs from group 3, generated by Tigress.

```
void SECRET(unsigned long input[1] , unsigned long output[1] )
{
  unsigned long state[1] ;
  char password[100]  = "";
  int failed  = 0;
  int stringCompareResult ;
  int activationCode ;
  unsigned long local1 ;
  unsigned long *output_ref  = output;
  char randomFunsSwapPartCopy21 ;
  char randomFunsSwapPartCopy22 ;
  char randomFunsSwapPartCopy23 ;
  char randomFunsSwapPartCopy24 ;
  unsigned short randomFunsSwapPartCopy26 ;
  unsigned int randomFunsSwapPartCopy27 ;
  unsigned short randomFunsSwapPartCopy29 ;
  unsigned short randomFunsSwapPartCopy32 ;
  char randomFunsSwapPartCopy33 ;
  char randomFunsSwapPartCopy34 ;
  unsigned int randomFunsSwapPartCopy37 ;
  char randomFunsSwapPartCopy39 ;
  unsigned short randomFunsSwapPartCopy40 ;
  char randomFunsSwapPartCopy41 ;
  unsigned int randomFunsSwapPartCopy42 ;
  char randomFunsSwapPartCopy43 ;
  char randomFunsSwapPartCopy44 ;
  unsigned int randomFunsSwapPartCopy45 ;
  unsigned short randomFunsSwapPartCopy46 ;

  {
  state[0UL] = (input[0UL] | 374233453UL) >> 1UL;
  if ((state[0UL] >> 4UL) & 1UL) {
    randomFunsSwapPartCopy21 = *((char *)(& state[0UL]) + 5);
    *((char *)(& state[0UL]) + 5) = *((char *)(& state[0UL]) +
3);
    *((char *)(& state[0UL]) + 3) = randomFunsSwapPartCopy21;
    randomFunsSwapPartCopy21 = *((char *)(& state[0UL]) + 7);
    *((char *)(& state[0UL]) + 7) = *((char *)(& state[0UL]) +
5);
```

Figure 19. Sample program generated by Tigress.

4.1.2 Obfuscations

We use obfuscators to apply various transformations in order to create a large set of obfuscated programs. The application of obfuscation is done in two sets. For the first, we apply only single transformations to the clean files. Every file is obfuscated with each transformation to produce a number of variants equal to the number of transformations; a

total of 8 variants per each original file. In addition to the 8 variants, some files were also obfuscated to produce a JIT variant; however, not every sample was compatible with the JIT transformation. This means that only some JIT transformed programs exist in our dataset.

Then, for the second set, we once again obfuscate the clean samples, but this time performing multiple transformation layered on top of each other. This produces a number of variants equal to the number of chosen permutations. No layered variants involving JIT were produced due to the complexity of the transformation.

```
10 #commands
11 A='--Transform=InitOpaque --Functions=main --Transform=AddOpaque --InitOpaqueStructs=list,array,input --Functions=main --AddOpaqueCount=16 --AddOpaqueKinds=call,bug,true,junk --Transform=CleanUp --CleanUpKinds=annotat
12 B='--Transform=EncodeLiterals --Functions=main --EncodeLiteralsKinds=integer,string'
13 C='--Transform=EncodeArithmetic --Functions=main'
14 D='--Transform=Flatten --FlattenDispatch=switch,goto --Functions=main'
15 E='--Transform=Virtualize --VirtualizeDispatch=switch,direct,ifnest,linear --Functions=main'
16
17
18
19 FILES=${1}/*.c
20
21 for FILENAME in $FILES
22 do
23         F=${FILENAME##*/}
24         FILE_W_EXT=${F%.*}
25
26         #AddOpaque, EncodeLiterals;
27         tigress --Environment=x86_64:Linux:Gcc:4.6 --FilePrefix=obf ${A} ${B} --out=virtmult/${FILE_W_EXT}-OL.c ${1}/${F}
28
29         #AddOpaque, EncodeArithmetics
30         tigress --Environment=x86_64:Linux:Gcc:4.6 --FilePrefix=obf ${A} ${C} --out=virtmult/${FILE_W_EXT}-OA.c ${1}/${F}
31
32         #EncodeArithmetics, AddOpaque
33         tigress --Environment=x86_64:Linux:Gcc:4.6 --FilePrefix=obf ${C} ${A} --out=virtmult/${FILE_W_EXT}-AO.c ${1}/${F}
34
35         #AddOpaque, EncodeArithmetics, EncodeLiterals
36         tigress --Environment=x86_64:Linux:Gcc:4.6 --FilePrefix=obf ${A} ${C} ${B} --out=virtmult/${FILE_W_EXT}-OAL.c ${1}/${F}
37
38         #EncodeArithmetics, Literals, Opaque
39         tigress --Environment=x86_64:Linux:Gcc:4.6 --FilePrefix=obf ${C} ${B} ${A} --out=virtmult/${FILE_W_EXT}-ALO.c ${1}/${F}
40
41         #Opaque, Flatten
42         tigress --Environment=x86_64:Linux:Gcc:4.6 --FilePrefix=obf ${A} ${D} --out=virtmult/${FILE_W_EXT}-OF.c ${1}/${F}
43
44         #Flatten, Opaque
45         tigress --Environment=x86_64:Linux:Gcc:4.6 --FilePrefix=obf ${D} ${A} --out=virtmult/${FILE_W_EXT}-FO.c ${1}/${F}
46
47         #Flatten, Arithmetic, Literal
48         tigress --Environment=x86_64:Linux:Gcc:4.6 --FilePrefix=obf ${D} ${C} ${B} --out=virtmult/${FILE_W_EXT}-FAL.c ${1}/${F}
49
50         #Virt, Opaque
51         tigress --Environment=x86_64:Linux:Gcc:4.6 --FilePrefix=obf ${E} ${A} --out=virtmult/${FILE_W_EXT}-VO.c ${1}/${F}
52
53         #Virt, Arithmetic, Literals
54         tigress --Environment=x86_64:Linux:Gcc:4.6 --FilePrefix=obf ${E} ${C} ${B} --out=virtmult/${FILE_W_EXT}-VAL.c ${1}/${F}
55
```

Figure 20. Partial view of the script use to produce obfuscated variants.

After producing the obfuscated variants with Tigress and OLLVM, our data set consists of over 100,000 programs split between both data sets. A list of the commands for our obfuscators and the permutations used for the multi layered samples will be made available.

4.2 Detection Suite

Prior work [14, 17, 18] has shown that the detection of obfuscation is possible, to the level of identifying the transformation in use, with machine learning. This phase will involve the implementation of a suite such detectors. This will allow us to gain an understanding of the current ability for obfuscation detection which will use a baseline for evaluating the effectiveness of our proposed defenses. Each of these detectors will be created differently, either relying on different machine learning models or feature set but will all be trained on the same dataset as outlined from earlier research [14]. Table 1 outlines our proposed detectors as well as the feature sets and models used. While some of our detectors will be based on related research, the rest will be created and implemented for our work. We will implement two new detectors based on gadget and image analysis, while also implementing a third based on opcode analysis from earlier work [14] but with a support vector machine as the underlying model.

Table 1. Description of Classifiers.

Feature Set	ML Models
Opcode TF-IDF	Decision Tree, Naïve Bayes, SVM
IR Symbolic Trace	Ensemble (Random Forest, Extra Tree)
Grayscale Binary Image	Convolutional Neural Network
Binary Gadgets	Decision Tree, Naïve Bayes, SVM

The purpose of these new models is not only to expand the range of our detectors, but also two determine the effectiveness of two new feature sets as well.

4.2.1 Gadget Based Detection

Gadgets are chains of opcodes within programs that can be taken advantage of by an attacker to cause a program to perform in a way other than intended even if the program is protected from code execution attacks [90]. The last opcode in a gadget is typically a return instruction that will be used to chain to the next gadget. As gadgets are based on the opcodes within a program, and the obfuscations chosen for our research are only those that would cause changes in opcodes, we reason that the programs in our obfuscated dataset would have differing gadgets from their original counterparts.

```
Gadgets information
============================================================
0x0000000000001460 : adc eax, 0xbe3 ; add rax, rdx ; jmp rax
0x0000000000001668 : adc eax, 0xc1b ; add rax, rdx ; jmp rax
0x0000000000001488 : adc eax, 0xd6b ; add rax, rdx ; jmp rax
0x0000000000001be5 : add al, 0x48 ; mov dword ptr [rbp - 0x1c8], eax ; jmp 0x1c21
0x00000000000014f3 : add al, 0x48 ; mov dword ptr [rbp - 0x1c8], eax ; jmp 0x1c21
0x0000000000001772 : add al, 0x48 ; mov dword ptr [rbp - 0x1c8], eax ; jmp 0x1c21
0x0000000000001a0b : add al, 0x48 ; mov dword ptr [rbp - 0x1c8], eax ; jmp 0x1c21
0x0000000000001b53 : add al, 0x48 ; mov dword ptr [rbp - 0x1c8], eax ; jmp 0x1c21
0x0000000000001464 : add byte ptr [rax + 1], cl ; sar byte ptr [rsi], 1 ; jmp rax
0x000000000000148c : add byte ptr [rax + 1], cl ; sar byte ptr [rsi], 1 ; jmp rax
0x000000000000166c : add byte ptr [rax + 1], cl ; sar byte ptr [rsi], 1 ; jmp rax
0x00000000000011bb : add byte ptr [rax], 0 ; add byte ptr [rax], al ; endbr64 ; jmp 0x1149
0x0000000000001133 : add byte ptr [rax], 0 ; add byte ptr [rax], al ; ret
0x00000000000011bc : add byte ptr [rax], al ; add byte ptr [rax], al ; endbr64 ; jmp 0x1148
0x0000000000001dfc : add byte ptr [rax], al ; add byte ptr [rax], al ; endbr64 ; ret
0x0000000000001263 : add byte ptr [rax], al ; add byte ptr [rax], al ; jmp 0x12a6
0x00000000000012c2 : add byte ptr [rax], al ; add byte ptr [rax], al ; jmp 0x12ff
0x000000000000131b : add byte ptr [rax], al ; add byte ptr [rax], al ; jmp 0x1352
0x0000000000001341 : add byte ptr [rax], al ; add byte ptr [rax], al ; jmp 0x1360
0x0000000000001cf4 : add byte ptr [rax], al ; add byte ptr [rax], al ; jmp 0x1d50
0x0000000000001134 : add byte ptr [rax], al ; add byte ptr [rax], al ; ret
0x00000000000018a1 : add byte ptr [rax], al ; add byte ptr [rdi], cl ; test byte ptr [rcx + 3], bh ; add byte ptr
[rax], al ; jmp 0x1c27
0x0000000000001036 : add byte ptr [rax], al ; add dl, dh ; jmp 0x1024
0x0000000000001046 : add byte ptr [rax], al ; add dl, dh ; jmp 0x1024
0x0000000000001056 : add byte ptr [rax], al ; add dl, dh ; jmp 0x1024
0x0000000000001066 : add byte ptr [rax], al ; add dl, dh ; jmp 0x1024
0x0000000000001076 : add byte ptr [rax], al ; add dl, dh ; jmp 0x1024
0x00000000000011b0 : add byte ptr [rax], al ; add dword ptr [rbp - 0x3d], ebx ; nop dword ptr [rax] ; ret
```

Figure 21. Example of Gadget list from a binary.

The feature set for this detector will be the extracted gadget list taken from a program. This detector will allow us to see if the distinct opcode changes introduced by transformation will produce similarly distinct changes in the gadgets that are present.

4.2.2 Image Based Detection

It has been shown that programs can be converted into images in order to be analyzed [40]. In fact, this technique has seen successful use in malware detection [40]. This process is commonly done by taking the bytes of a program and having each of those bytes represent a pixel in the formed image, producing a grayscale image that is a representation of the original program [40]. Colored images can be produced by having 3 bytes of program represent the RGB values of a single pixel.

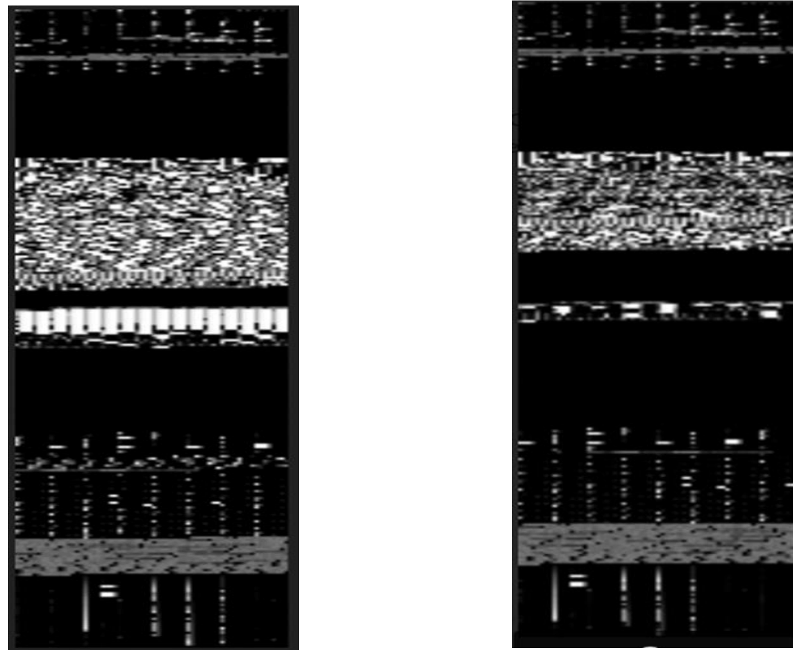

Figure 22. Two examples of a binary converted to a Grayscale image.

We reason that since this technique has been successfully use for malware detection, it can be used to identify obfuscations as well. Transformations alter the opcode layout of a program in noticeable ways which we believe will be reflected in the resulting image.

4.3 Evasion of Detectors

In the second phase, we will begin focusing on how we may modify obfuscated samples in order to increase their stealth. In our context, we informally define stealth as a samples ability to avoid detection by a machine learning based detector. This would mean that decreases in the classification rate of our detectors equals an increase in the stealth of our samples. From an adversarial view, increases in the evasion rate of our samples equals an increase in the stealth of our samples. To accomplish this goal, we propose and will explore two methods: Adversarial Obfuscation and Obfuscation Expansion. We will determine the effectiveness of these methods by analyze the impact they have on the baseline performance of our detectors determined from the previous phase.

4.3.1 Adversarial Obfuscation

Adversarial Obfuscation is what we are calling the process of transforming obfuscated programs into adversarial examples made to evade detection by a machine learning based detector. This is achieved by the application of Adversarial ML attacks where we are acting as the adversary and the detectors are the models being attacked. Due to this, this defense method follows heavily the general process outlined in earlier

sections for creating adversarial samples. Some changes are required however for our purposes. Figure 21 is a visual outline of this process.

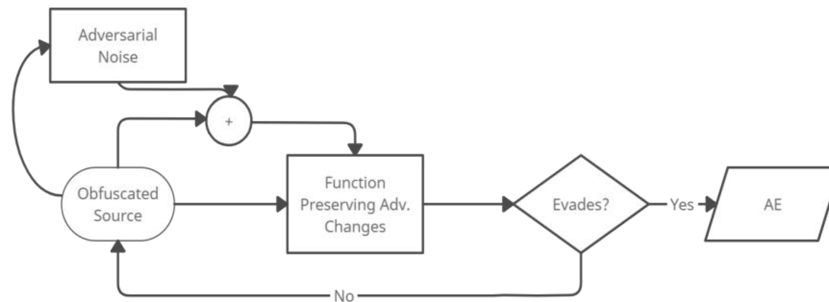

Figure 23. Adversarial Obfuscation Process.

These changes are in relation to the requirements that after becoming adversarial samples, the programs must still be functioning, and the obfuscating transformation must remain intact with minimal negative impact to metrics. The first of these requirements have been encountered before with the use of Adversarial ML to construct malware samples that could evade detection [91, 92, 93]. This shows that there are workarounds for this problem. The second constraint will require us to test the generated samples potency, resilience, and cost against those of the original to see the impact the changes have had [94, 28]. Using this, we will maximize our evasion potential while minimizing the impact to metrics to produce adversarial samples that are functioning, evasive, and obfuscated.

4.3.2 Obfuscation Expansion

We are calling our second proposed defense method Obfuscation Expansion. In this method we will focus our analysis on identifying the features present in samples that

result in their classification by detectors and then minimizing the uniqueness of those features within the sample by “expanding” the code base. Figure 22 provides a visual.

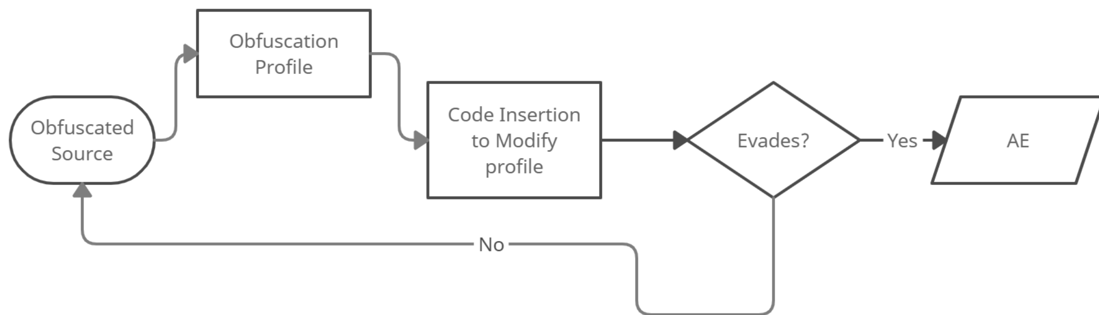

Figure 24. Overview of Code Expansion Process.

We began with a program P that has been previously obfuscated. P has a set of set of features F that will be extracted and examined by the detectors to classify the transformation T used on P. This done using the relations the detector has determined between the individual features in F and their relations to different T’s. In other words, if the features in F from P relating to a given T are uniquely expressed from all other features in F, P is considered to have been transformed using T. With this in mind, we will expand the code base of P by introducing code that relates to features not associated with the actual T using on the sample. This will mean that the features connected to T are no longer unique within F. We believe this will result in misclassification.

In this research, we accomplish this by adding additional opcodes to the original program. Even though two of our detectors are not based directly on opcode analysis, both of their feature sets are derived from opcodes and therefore we believe they should be affected by this method as well. We expect this method to increase the cost overhead

of a transformation but with little to no impact on potency or resilience.

4.3.3 Impact on Obfuscation

After samples have successfully evaded detection by one or more of our detectors, we will analyze the samples to test for the impact our defenses have had on the obfuscation present within the original samples. Since our proposed methods are either modifying or adding to the existing code, we can expect this to impact the potency, cost, and resilience of the transformation in some way. We will test our samples using established ways to obtain these metrics to gain an understanding of these impacts [28, 94]. This will allow us to refine our defenses to maximize the increase in stealth while minimizing the decreases to other metrics.

4.4 Automation and Comparison

Once the evasion experiments have been completed and our two proposed modification methods can be properly applied to samples, we will construct a tool or toolchain that will automate the process of creating samples with our modifications applied. Ideally, we will construct a standalone tool capable of taking in an original program that it will then obfuscate while applying either the Adversarial Obfuscation or Obfuscation Expansion defenses at the choice of the user. If constraints prevent the development of a standalone tool, we will instead create a toolchain that accomplishes the same process using existing tools driven by software we create. There are two purposes behind the automation of our defenses, enhanced speed of sample production and to aid in additional research.

By automating the process of sample production outlined in phase 2, we will be

able to produce many samples in a shorter period of time. This will enable to perform more analysis on the impact our proposed defenses have on existing obfuscation across a broader degree of programs. This will allow further fine tuning of our defenses. The greater the number of samples we are able to analyze will also allow us to perform comparisons of our two defenses, which will give us the answer to one of our research questions. We will compare Adversarial Obfuscation and Obfuscation Expansion in terms of evasion rate, potency impact, resiliency impact, and cost overheard in order to gain an understanding of one method's benefits over the other. If it exists, we will also compare error rate, the rate at which our tool produces nonfunctioning samples of either type.

The second benefit of automation is that it will make future research in this area, either adjacent or furthering, easier to perform as the production of samples will not be left wholly up the researchers. We believe that at the completion of this research there will still be more avenues to explore so automation will allow faster exploration of those new research threads.

CHAPTER V

OBFUSCATION CLASSIFICATION

The following chapter details our research in using machine learning to perform automated metadata recovery of obfuscated programs. We develop and evaluate a variety of classifiers based on three different categories of features extracted from source programs. Parts of the content of this section are taken from the paper "Machine learning classification of obfuscation using Image Visualization" which was published in the *Proceedings of the 18th International Conference on Security and Cryptography* [96].

5.1 Introduction

This research explores three types of features that can be extracted from obfuscated programs for the purpose of training machine learning classification models. These models can then be used to perform automated metadata recovery attacks, which can give an adversary information about the type of obfuscation in place on program. The feature sets we explore are:

- Opcode N-Grams: opcode sequences of varying length, taken directly from the assembly code of a program.
- Gadgets: Sequences of opcodes used in Return-Oriented Programing attacks.

- Code Images: Grayscale images created by using a program's bytes as pixel values.

We train and evaluate our models on a corpus of obfuscated programs, made using the Tigress and OLLVM obfuscators. Our tests frame this as both a binary problem through the use of many individual models as well as a multi label problem via a single model. Both approaches are shown to be highly effective at classifying a transformation in a program, even in the presence of multiple transformations in a layered fashion.

Our results show that all feature sets can produce models with an average f1-score over %95 at identifying a single obfuscating transformation present in a file. Further, opcode n-grams and gadget lists produce f1-scores at over %96 at labeling transforms that are either layered or present at different points in a program. Our feature sets are also used to produce models that can identify fine-grained features of an obfuscating transformation. We make the following contributions with this research:

- The use of image analysis via code visualization and convolutional neural networks as an avenue for performing metadata recovery attacks on obfuscated programs. Code visualization allows for features of a program to be analyzed without any reverse engineering of the program.
- We evaluate the effectiveness of supervised learning models trained on images of obfuscated programs at classifying the transformation in use on previously unseen samples. The evaluations are performed with both binary and multi label classification models. Both approaches show a high accuracy across a range of transformations.

- We demonstrate that the analysis of obfuscated images has potential for higher granularity by evaluating against samples with transformations layered one after the other. High accuracy is maintained for these samples, despite the increase in complexity.
- We evaluate the use of extracted gadget lists as a means of training supervised learning models for the purposing of identifying transformations present in binaries.

5.2 Background

In this section, we briefly describe the basics of a convolutional neural network, which will be used in the methodology of this chapter.

5.2.1 Convolutional Neural Networks

When using images as input for a classification task it is important to properly capture the spatial relationships of the pixels in the image [97]. While there are many ways to achieve this, Convolutional Neural Networks (CNN) have become a popular choice. Designed with images in mind, CNN are capable of learning from an image in pieces in order to understand the whole [98]. This is done with one or more convolutional layers. These layers learn move through sections of the image, learning a representation of that section. Convolutional layers are followed by pooling layers, and CNN are then comprised of one or more dense layers as is common of Deep neural networks. The use of CNN for analyzing images led to them being applied for tasks such as malware detection [99, 100].

5.3 Methodology

In this section, we will introduce and walkthrough the methodology for this research. An outline of our methodology can be seen in Figure 25.

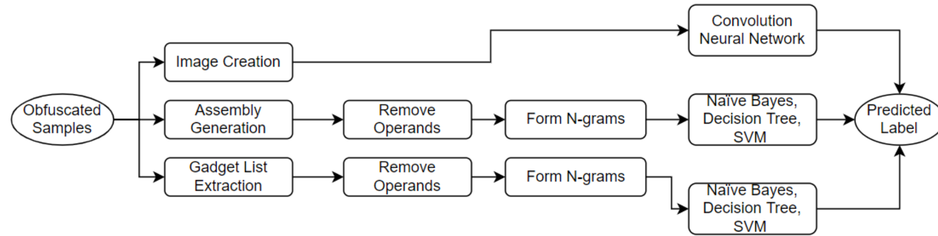

Figure 25. Outline of Methodology.

We must first form the feature sets that will be used to train and test our models. This will be for each of the three feature types: images, n-grams, and gadgets. After extracting these sets, we then use them to train various models to label transformations within programs. For training the models, we will make use of 10-fold cross validation and also a variant known as functional cross variation. This difference between functional and regular cross validation lies in the forming of the train and validation sets. In the functional variant, no version of a sample can be a part of the validation set if another version of that sample appears in the training set. All versions of a sample must appear in the training or validation set.

5.3.1 Dataset creation

The dataset used in this research is described in Chapter IV. For the purpose of our supervised learning, the labels used for the samples will be the obfuscating

transformations used on the programs. The following sections describe each of the feature extractions.

<u>5.3.1.1 Image Creation.</u> The basics of creating grayscale images from programs was outlined in Chapter IV. When creating the code images, we have two versions of each sample. Both sets are square images, with the height and width of the first being determined based on the size of the program. The second set is generated with a width of 256 and a computed height. The height is then padded in order to produce 256 x 256 images of each program.

<u>5.3.1.2 Disassembly.</u> For our opcode n-gram feature sets, we need to obtain assembly code from the c-source files. The typical way an adversary who does not have access to source code would obtain this is by using a disassembler on the compiled program. This would give them an assembly representation of the program, with perhaps a few errors. While going this route would be the most challenging for out model and a more real world setting for a malicious actor creating the model, we chose instead to get a programs true assembly representation by having it outputted directly from the gcc compiler. This can be done by using the "-S" flag and will cause the source file to go through preprocessing and initial compilation but will stop before the assembler is ran.

We do this so that the models trained on this feature set are representative of a worst-case scenario where an adversary has perfect or near perfect assembly output. This is done since prior work has already explored using disassembly to train models for labeling obfuscations, and so that this model will be more resistant to the detection avoidance techniques employed later in this research. Figure 26 shows an example of disassembly output.

```
main:
        pushq   %rbp
        .seh_pushreg    %rbp
        movq    %rsp, %rbp
        .seh_setframe   %rbp, 0
        subq    $64, %rsp
        .seh_stackalloc 64
        .seh_endprologue
        movl    %ecx, 16(%rbp)
        movq    %rdx, 24(%rbp)
        movq    %r8, 32(%rbp)
        call    __main
        call    obfmegaInit
        movl    16(%rbp), %eax
        movl    %eax, _global_obfargc(%rip)
        movq    24(%rbp), %rax
        movq    %rax, _global_obfargv(%rip)
        movq    32(%rbp), %rax
        movq    %rax, _global_obfenvp(%rip)
        movl    $1, -4(%rbp)
        movl    16(%rbp), %eax
```

Figure 26. Sample assembly output.

5.3.1.3 Gadget Extraction. The final feature set is the gadget lists. The basics of gadgets were explained in Chapter IV. The gadgets were extracted using the tool ROPGadget [101]. Similar to the image feature set, there are two variants of each gadget list. ROPGadget has option that removes duplicate gadgets from the list, leaving only one gadget of each sequence of opcodes. We run ROPGadget both with and without this option, to see the impacts of duplicates on the models during training and testing.

5.3.2 Classifiers

This section details the classifiers used in this work and the feature sets used to train each classifier.

5.3.2.1 CNN. For the supervised learning model developed from our image dataset, we choose to use a CNN, as high accuracy has been observed from CNN when classifying images made from malware samples. For our model architecture, we choose to use a small model consisting of four convolutional layers which feed into two dense layers. This is because models with this architecture and others similar to it have been shown to be proficient at classifying the Malimg data set without requiring high degree of resources. The specifics of our model can be seen in Fig. 27. Our input layer is shaped to take in the pixel values of our images directly as opposed to extracting some feature or aspect of the whole image. This is to see how much information can be obtained without a high degree of preprocessing or prior analysis.

5.3.2.2 FCNN. Since programs can come in a wide range of sizes, we train and test a Fully Convolutional Neural Network (FCNN). This FCNN is created by reimplementing all the dense layers of our CNN as convolutional layers, with a number of filters equal to the nodes in the dense layer and featuring 1x1 convolutions. The specifics of this model can be seen in Figure 28. This style of model is commonly used in image segmentation tasks and allows to process models of varying sizes, without any modifications.

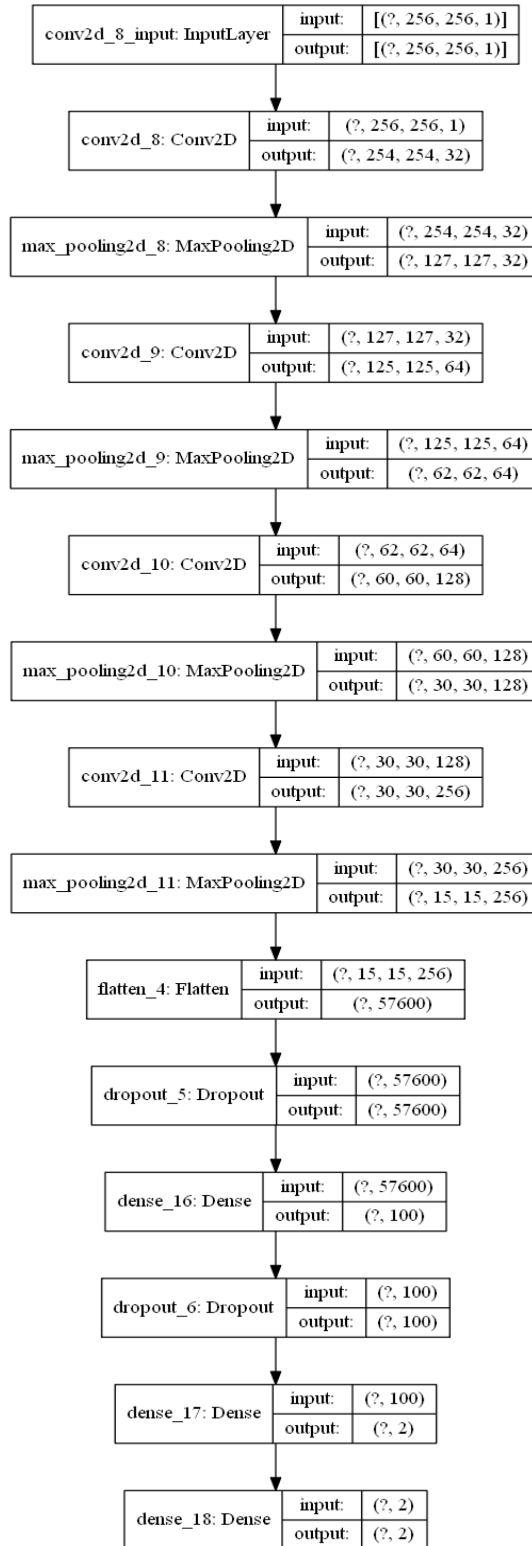

Figure 27. Figure showing CNN architecture.

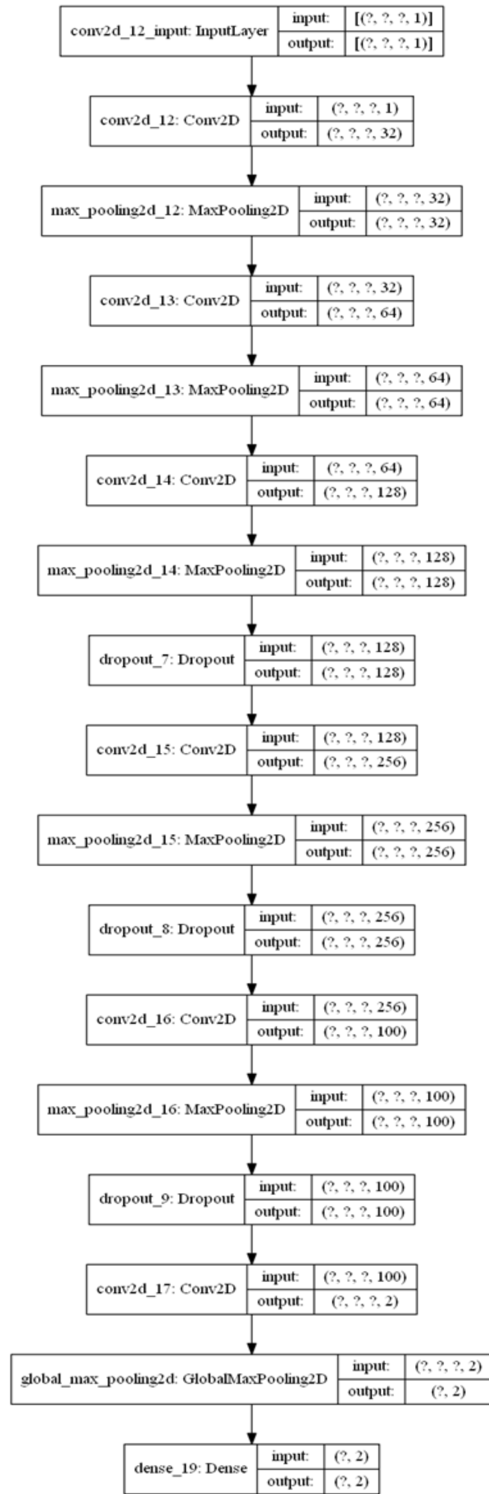

Figure 28. Fully Convolutional Neural Network Architecture.

5.3.2.3 Opcode. For our opcode analysis, we construct three different types of classifiers: Naïve Bayes (NB), Decision Tree (DT), and Support Vector Machine (SVM). All of these models will be trained using the disassembly files as their dataset; with the feature set being extracted opcode n-grams. An opcode n-gram is a sequence of adjacent opcodes of length *n*. For this research, we extract three sets of n-grams of increasing lengths. The first set consists only of 1-gram sequences; just normal opcodes. The second includes the first set, as well as 2-gram sequences. The final set combines the first two and adds in 3-gram sequences. We remove the operands and leave only the opcodes. For example, given the disassembly [add, mov, jp, xchg, sub, ret]:

- 1st set: add, mov, jp, xchg, sub, ret.
- 2nd set: add mov, mov jp, jp xchg, xchg sub, sub ret, add, mov, jp, xchg, sub, ret.
- 3rd set: add mov jp, mov jp xchg, jp xchg sub, xchg sub ret, add mov, mov jp, jp xchg, xchg sub, sub ret, add, mov, jp, xchg, sub, ret.

While n-grams will be used as our features, Term Frequency-Inverse Document Frequency (TFIDF) will be used as the feature values. TFIDF can be calculated as follows:

$$tfidf(w, d, D) = tf(w, d) * idf(w, D)$$

$$tf(w, d) = log\ (1 + f(w, d))$$

$$idf(w, D) = \log\left(\frac{N}{f(w, D)}\right)$$

With *w* being the n-gram, *d* being a single document, *D* being the entire dataset, *N* being the number of samples in the dataset, and the functions being the frequency of the

n-gram within the sample and a dataset. Our three classifiers will each have three variants, based on the feature set used for training. This is done to examine the importance of sequence length to metadata recovery. As a final step before training, feature selection is performed using chi-squared [102]. Anything with a p-value below 95% is removed from the feature set.

5.3.2.4 Gadgets. Our gadget dataset will use the same classification algorithms as our disassembly. This is due to the fact that gadgets can be viewed as a type of opcode sequence. Since the gadgets are in order based on the location of the first opcode in the sequence, we still use the same n-gram sets as discussed previously. For example, given the gadgets [add add jmp, add jmp, add test je call] our sets would be:

- 1st: add add jmp, add jmp, add test je call
- 2nd: (add add jmp, add jmp), (add jmp, add test je call), add add jmp, add jmp, add test je call
- 3rd: (add add jmp, add jmp, add test je call), (add add jmp, add jmp), (add jmp, add test je call), add add jmp, add jmp, add test je call

Much like with the disassembly, we remove the operands and leave only the opcodes in the gadget sequence.

5.4 Results

This section details the results of the methodology described in the previous section. Each of our models will be trained and evaluated using functional cross validation. The reported value for testing is the models *f1*-score.

5.4.1 CNN

Table 2 shows the results of our CNN classification test. The results are separated based on the number of layers present in the samples used for training and testing. This was done so to examine the impact of layering on our image analysis and to see what the changes in accuracy were across the range of transformation during layering. This model was trained and tested on the images that were padded to 256x256 squares.

Table 2. F1-scores of CNN model at differing layers.

	1-layer	*2-layer*	*3-layer*	*4-layer*	*5-layer*
Flatten	99.5	98.2	95.7	91.1	85.4
Virtualize	100	99.2	95.2	92.7	85.7
Encode L.	99.1	97.3	90.6	83.4	75.3
Encode A.	99.7	97.9	94.1	90.7	86.9
Opaque	99.6	96.4	89.3	82.8	75.4

5.4.2 FCNN

Table 3 shows the results of our FCNN training and testing. The training setup for this network was identical to the previous network. The only difference is with the training and test data. This model was trained on the images produced at algorithmically determined square sizes.

5.4.3 Opcode

For our opcode-based classifiers, we train three groups of classifiers with each group having three variants of the same type of model. The three variants are the ones based on the different lengths of n-grams discussed previously. The three models in question are decision trees (specifically a Gini tree), naïve bayes, and a support vector

machine. Tables 4, 5 and 6 show the different classification results for the single layer samples, while Tables 7, 8, and 9 show the results for the multi-layer samples.

Table 3. Classification f1-scores for fully convolutional model.

	1-layer	*2-layer*	*3-layer*	*4-layer*	*5-layer*
Flatten	70.2	68.3	67.2	63.5	69.3
Virtualize	72.4	68.3	64.2	62.9	61.8
Encode L.	73.9	68.3	66.2	63.2	62.1
Encode A.	78.2	71.2	67.5	65.7	65.4
Opaque	69.1	69.5	67.1	64.1	59.2

Table 4. Naive Bayes single layer results.

	1-GRAM	**1,2-GRAM**	**1,2,3-GRAM**
FLATTEN	98	99	99
VIRTUALIZE	99	100	100
ENCODE L.	98	100	98
ENCODE A.	100	99	97
OPAQUE	95	93	90

Table 5. Naive Bayes multi-layer results.

	1-GRAM	**1,2-GRAM**	**1,2,3-GRAM**
FLATTEN	50	47	48
VIRTUALIZE	86	86	83
ENCODE L.	72	74	71
ENCODE A.	83	84	83
OPAQUE	50	47	48

Table 6. Decision Tree single layer results.

	1-GRAM	1,2-GRAM	1,2,3-GRAM
FLATTEN	99	99	99
VIRTUALIZE	99	100	100
ENCODE L.	99	99	99
ENCODE A.	99	99	99
OPAQUE	99	100	99

Table 7. Decision Tree multi-layer results.

	1-GRAM	1,2-GRAM	1,2,3-GRAM
FLATTEN	98	98	98
VIRTUALIZE	99	99	99
ENCODE L.	98	98	98
ENCODE A.	99	99	99
OPAQUE	99	99	99

Table 8. SVM single layer results.

	1-GRAM	1,2-GRAM	1,2,3-GRAM
FLATTEN	100	100	100
VIRTUALIZE	100	100	100
ENCODE L.	100	100	99
ENCODE A.	100	100	100
OPAQUE	99	100	99

Table 9. SVM multi-layer results.

	1-GRAM	1,2-GRAM	1,2,3-GRAM
FLATTEN	99	100	99
VIRTUALIZE	99	100	99
ENCODE L.	98	100	98
ENCODE A.	99	100	99
OPAQUE	98	100	98

5.4.4 Gadgets

Tables 10, 11, and 12 show the classification *f1*-scores for our models trained and tested on gadgets lists from our obfuscated samples. These models were formed the same as the opcode n-gram models.

Table 10. Naïve Bayes single layer results.

	1-GRAM	1,2-GRAM	1,2,3-GRAM
FLATTEN	99	100	99
VIRTUALIZE	100	100	100
ENCODE L.	100	100	99
ENCODE A.	99	99	99
OPAQUE	99	99	99

Table 11. Naive Bayes multi-layer results.

	1-GRAM	1,2-GRAM	1,2,3-GRAM
FLATTEN	97	96	99
VIRTUALIZE	100	100	100
ENCODE L.	98	97	97
ENCODE A.	97	95	96
OPAQUE	93	91	91

Table 12. Decision Tree single layer results.

	1-GRAM	1,2-GRAM	1,2,3-GRAM
FLATTEN	99	99	99
VIRTUALIZE	99	100	100
ENCODE L.	99	99	100
ENCODE A.	99	100	100
OPAQUE	99	99	99

Table 13. Decision Tree multi-layer results.

	1-GRAM	1,2-GRAM	1,2,3-GRAM
FLATTEN	99	99	99
VIRTUALIZE	99	100	100
ENCODE L.	99	100	99
ENCODE A.	99	99	99
OPAQUE	99	99	99

Table 14. SVM single layer results.

	1-GRAM	1,2-GRAM	1,2,3-GRAM
FLATTEN	99	99	99
VIRTUALIZE	99	100	99
ENCODE L.	99	99	99
ENCODE A.	100	100	100
OPAQUE	99	99	99

Table 15. SVM multi-layer results.

	1-GRAM	1,2-GRAM	1,2,3-GRAM
FLATTEN	99	100	99
VIRTUALIZE	99	100	98
ENCODE L.	98	100	99
ENCODE A.	98	100	99
OPAQUE	99	100	99

5.4 Discussion

We will discuss our groups of models in the order that they were tested. Our CNN model was shown to be highly accurate on single- and two-layer samples, scoring over 97% for each transformation. However, beginning with the 3-layer samples, we started to see a drop in performance. This is most likely due to the transformations serving to hide each other from the type of analysis being performed. The most recent transformations

are obscuring the ones that were applied first, meaning that the information was not extracted via the pixel values. We can notice with these results however that we can gain a glimpse of which transformations are ‘heavy’ or that are harder to hide. The transformations that dropped in classification performance faster are lighter transformations. Our other CNN, the FCNN, did not successfully trained and can be viewed as a failure.

The second group of classifiers, the opcode-based models, performed above expectations. Both the decision tree and svm model sets-maintained scores above 98% across all n-gram sizes and independent of multi or single layer samples. The only model to underperform is the Naïve Bayes model. While it kept pace with the other models in the single layer, it was unable to properly classify multi-layer samples.

The last model set, the gadget-based models, was the highest performing set. Each model was capable of reaching a score of 100% for certain transformations in both single and multi-layered settings.

When compared to related work discussed in Chapter III, our models performed comparably. This shows that we can be confident in moving forward with these models in upcoming work.

5.5 Future Work

Our current course with this work falls along 2 paths: feature set exploration/refinement and increasing the granularity of our ML models. In combination with related work, five feature sets have been shown capable of training models for automated metadata recovery [14,17,18]. Exploration of other feature sets will broaden

range of models that can created for analyzing software. More feature sets also allow for models that can cover gaps and capabilities presented by other models. This will help us to further understand the level of metadata that can be automatically extracted.

The second path is concerned with increased granularity of our attacks. In this work, we showcased the ability to determine the types of transformations used on a file, as well as certain information about that transformation. Further increasing the granularity could potentially allow for all aspects of a transformation to be learned, giving analysts a much greater ability to perform deobfuscation tasks. Our immediate goal on this path is the ability to label the bytes of a program that are part of a given transformation.

5.6 Conclusion

In this section, we have proposed and evaluated methods for automated labeling of obfuscating transformations applied to software, via three unique feature sets: Code visualization, Opcode N-grams, and Gadgets. A range of supervised learning models produced using these sets were able properly perform metadata recovery attacks and identify the types of transformations that had been applied to the file. For code visualization, our models produced *f1*-scores above 96% when presented with files containing only a single transformation. While the models scores did begin to drop as layered transformations were introduced into training and testing, most transformations could be identified with a score over 90% in the presence of four transformations layered over each other.

Models produced on our other two feature sets outperformed our image-based models in our various tests. Both opcode and gadget-based models were able to identify single transformations at 98%, with layered transformations only dropping to 95% across all transformations. This shows that gadget analysis is an effective feature set for obfuscation analysis along with opcodes and images.

CHAPTER VI

EVADING OBFUSCATION CLASSIFICATION

The sections in this chapter detail our efforts to improve the stealth of obfuscating transformations in order to deter automated metadata recovery attacks.

6.1 Adversarial Machine Learning

The research in this section makes use of adversarial machine learning in order to modify obfuscated programs into functional adversarial examples (AE). These modified programs are harder to classify for certain types of models without requiring extensive modification of the program.

6.1.1 Introduction

This research introduces and explores the process of Adversarial Obfuscation; a method of creating functional adversarial examples from obfuscated programs and AEs generated from adversarial machine learning algorithms. In the same vein as adversarial attacks in image analysis, AEs of obfuscated programs are executable binaries that have been modified with the purpose of evading a supervised learning model. Unlike natural images, these binaries cannot be changed freely, as they must preserve the original functionality of the program.

We present a process of creating executable adversarial examples from obfuscated binaries, by making minimal modifications based on computed distance between the original and a non-executable AE. This approach is extended from related work target at crafting AEs from malware binaries, in order to be used in the multi-class/multi-label space of obfuscation detection. We dub this modified approach as "Adversarial Obfuscation."

We report the misclassification rate of our adversarial obfuscated binaries against a suite of supervised learning classifiers, many of which were shown in Chapter V. We craft and test our samples using both single and multi-layered obfuscation samples.

The main contributions of this work are as follows:

- We present an approach for generating adversarial obfuscated binaries and for algorithmic dead code insertion using code images.
- We show that our adversarial obfuscated binaries are effective in black box attacks via adversarial transference.
- We explore the effectiveness our method against classifiers that do not make use of image analysis.
- We compare samples generated using our method against the approaches of instruction substitution and random dead code insertion.

6.1.2 Executable Adversarial Examples

Unlike natural images, there is an additional hurdle to consider to when creating adversarial examples (AE) from programs; the resultant AE must remain a functional program that is semantically equivalent to the original. This means that the noise introduced by adversarial ML algorithms cannot be used as is and must be modified or

adapted in some way, in order to be usable. The modifications however must still enable to program to function as an AE. This problem has been explored before in research focusing on the creation of AE for malware [103 - 105]. Some methods focus on limiting the changes to parts of the program outside of the .text section. In other words, the non-executable parts of the program. The problem with this method is that it is ineffective on any analysis that only focuses on executable portions. For example, if a model's preprocess involves the removal of the program header or the .data section.

Another approach that has seen success is inserting semantic NOPs into a program in order to form an AE [106]. This method involves first creating a non-functioning AE and measuring the distance between the original and the AE. Semantic NOPs are then inserted to minimize the distance between the original and the AE. Samples produced in this way have been shown to be effective and the changes are not limited to certain portions of the program. This is the approach we will be taking in this work.

```
1 xor eax, eax
2 mov eax, 0x45
3 mov ecx, 0x20
4 ret
```

```
1 xor eax, eax
2 nop
3 mov eax, 0x45
4 mov ecx, 0x20
5 ret
```

Figure 29. Basic nop insertion.

6.1.3 Methodology

This section explains the methodology used in this portion of our research. We continue to make use of the same dataset introduced and used in the previous chapters. We also use the classifiers created in Chapter V as the targets for our evasion attacks. We begin by taking an obfuscated program whose transformation(s) can be correctly labeled by our classifiers. We then put this program through the process of Adversarial Obfuscation. An overview of this process can be seen in Figure 30.

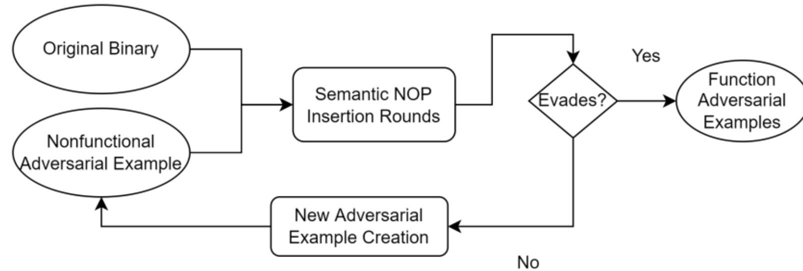

Figure 30. Adversarial Obfuscation overview.

We begin by transforming our sample into an image and producing an AE using the Fast Gradient Sign (FGSM) and Carlini-Wagner (CW) methods. We use our binary neural networks in order to perform these attacks. After the AE is created, we measure the distance between our original and the AE. The original sample then goes through a round of guided insertion of semantic NOPs. The goal of these insertions is to reduce a distance metric and bring the original sample closer to the AE. If at the end of the insertion round, the original sample is still not sufficient as an AE, a new AE is generated by CW using the modified sample and another round of insertion begins. After the final

round of the process, the original sample will have been modified into a semantic preserved AE.

In addition, we also perform distance comparisons on variants of the same sample produced by different obfuscating transformations. This is to explore the possibility of generating an AE using a different variant instead of a non-functioning AE.

6.1.3.1 Distance Comparisons. We will measure the distance both between the various transformations, as well as between the function and nonfunctional examples during adversarial obfuscation. Distance can be viewed as the measure of similarity between two points. To measure distance, we explore the use of four different equations: L1-norm (Manhattan distance), L2-norm (Euclidean distance), structural similarity image measure (SSIM), and binary distance (BD). These equations can be seen below. The L1 and L2 norms are common distance metrics, and the L2-norm will be used as part of our CW attack to generate AEs.

$$L1(a,b) = |a_1 - b_1| + \cdots + |a_n - b_n|$$

$$L2(a,b) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + \cdots + (a_n - b_n)^2}$$

$$SSIM(x,y) = \frac{(2\mu_x\mu_y + c_1)(\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)}$$

$$BD(x,y) = \sum_{i=0}^{n} \begin{cases} 0 & x[i] = y[i] \\ 1 & x[i] \neq y[i] \end{cases}$$

SSIM was created as a method for predicting the perceived quality of digital television. It measures the amount of noise between two images and reports that as

similarity. We treat this similarity as distance. Binary distance is used specifically for measuring the distance between two binary vectors. Each bit of the vectors is compared, if they are different then the distance is increased by one. It should be noted that all of the given distance metrics require the objects being measured to be of equal size or length.

6.1.3.2 Adversarial Obfuscation. The following section details the Adversarial Obfuscation process. This process is used to modify obfuscated binaries into functional adversarial examples.

6.1.3.2.1 Generating adversarial examples is the first step in the process. We generate our adversarial examples using FGSM and CW using the binary neural networks created in Chapter V as the targets. We use the binary neural networks as it makes creating samples for the multi-labeled programs significantly easier. We begin by applying multiple passes of FGSM to the code image. This is done as FGSM is not a computationally intensive process. This allows us to quickly generate an AE in order to significantly drop accuracy.

After FGSM has been applied, we use the *L2*-attack of CW to achieve the desired evasion rate for the AE. The CW attack can make more subtle and impactful changes to the image, but it is much slower and more computationally heavy. This is why it is done after multiple rounds of the. FGSM This gives us our final non-functioning AE.

This AE is then compared to our original binary using the binary distance metric. With both transformed to binary vectors, we begin traversing through the original binary. Making use of a list of insertion points, locations where a semantic nop can be inserted without changing the original semantics, at each point the algorithm compares the change in distance caused by inserting the various semantic nops. Distance is measured from the

current insertion point to the end of the binaries. The nop that has the greatest reduction in distance is inserted and the algorithm continues. If no nop reduces distance, then nothing is inserted. As nops our inserted, our original binary will be longer than our adversarial example. The binary will be padded at the front with zeros, as padding the end would impact the distance measure.

At the end of this process, the modified binary is put through the original classifier. If the sample is misclassified, then the algorithm is a success. In the event that the sample is still not misclassified or a desired probability value from the model is not achieved, the modified exampled can be used to create a new AE using Carlini-Wagner and the process will begin again. We call the process Adversarial Obfuscation (AO) and introduce it as a variant of Adversarial Malware Alignment Obfuscation (AMAO) [106].

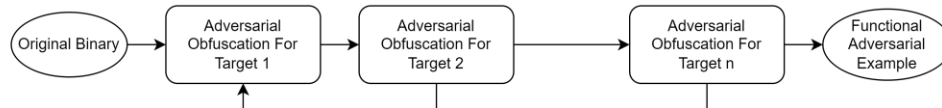

Figure 31. Waterfall version of Adversarial Obfuscation. n = the number of obfuscations or target obfuscations.

In the event that the binary possesses multiple obfuscations, or a targeted attack is desired (making the binary appear to have a transformation it does not), the waterfall variant of AO is used. This a lengthier process that sees the sample going through the traditional AO process with each label, being fed back into the previous classifiers after each successful completion. By the end of the process, it will achieve the desired labels as an adversarial example.

6.1.3.2.2 Semantic nops, also called dummy or dead code, are code sequences that do not affect the program logic. These sequences are named after the nop opcode. These nops can be inserted into a program to modify the makeup of the program in a semantic preserving fashion. Nops can vary in length but in this work, we deal mainly with nops consisting mainly of 1 – 2 instructions. Our list of nops can be seen in Table 16 [107].

Table 16. Semantic NOPs.

mov edi, edi	mov bx, bx	xchg ecx, ecx	add 0, rax
xchg ebx, ebx	xchg bx, bx	xchg edx, edx	and eax, eax
xchg cx, cx	sub 0, eax	mov bl, bl	and edi, edi
push rax; pop rax	mov esi, esi	nop	nop DWORD PTR [eax]
push rbx; pop rbx	mov al, al	xchg ax, ax	nop DWORD PTR [eax+eax*1+0x0]
mov ax, ax	push rcx; pop rcx	mov cx, cx	nop DWORD PTR [eax+0x0]

6.1.3.3 Using on Each Classifier. To test the effectiveness of our generated samples, we attempt to classify them with a variety of models. We evaluate in both a white-box and black-box setting. In our white box attack, we use a new CNN capable of classifying the samples as the means to generate the AE. This CNN is shown in Figure 32. After going through AO the samples are fed back into the same model.

In our black-box attack, we attempt to classify the samples with new CNNs. This would constitute a substitution attack. Our CNNs for this attack consist of one new CNN, the CNN from the previous chapter, and the InceptionV3 neural network [108]. InceptionV3 is included to test against a large and robust network.

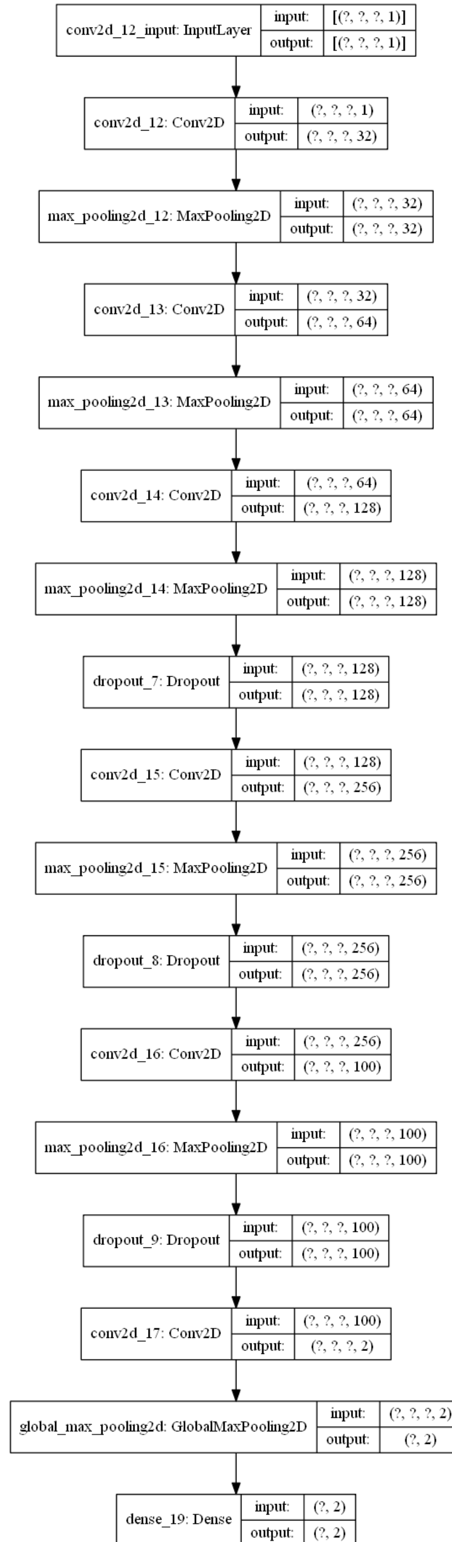

Figure 32. Architecture of CNN used to generate AEs.

We also classify the samples with the opcode n-gram and gadget classifiers from the last section as well. We do not expect the samples to be very effective against these models, but they are included to examine the overall effectiveness of our approach. For these models, we choose to test against only the 1,2-gram multi-layer models.

6.1.4 Results

The following tables showcase the results of the experiments performed in this section. For this test, we randomly selected obfuscated samples, both single and multi-layered, to be used in the creation of our adversarial examples. We begin by generating these samples for a nontargeted attack. Table 17 shows the classification results against our models, with the score for random insertion being presented as range, due to the attack being carried out multiple times. For multilayer samples, we make use of the waterfall model to evade all relevant classifiers. For the CNN score, this is an average obtained from all of the layered models. Only the model labeled CNN1 is a white box attack as this model was used to generate the samples.

Table 18 shows the results of applying our waterfall model to produce targeted attacks. We perform these attacks on both single and multi-layer samples. We generate samples to have up to 5 targets. We carried out this attack against CNN 2 and CNN 3.

6.1.5 Discussion

Our results show that adversarial examples created from our Adversarial Obfuscation approach are capable of successfully evading obfuscation detection systems based on image analysis. Our generated samples reduced the f1-scores of three Convolutional neural networks with different architectures to 0 and impacted the score of the InceptionV3 model. Our approach was also effective at creating examples for targeted

attacks. The same level of impact was achieved and was maintained even when a sample was made to target multiple classifiers.

Table 17. Classification results for Adversarial Obfuscation.

	Original	*Adversarial Obfuscation*	*Random Insertion*
CNN 1	90.8%	0%	25-35%
CNN 2	89.4%	0%	25-35%
CNN 3	89.7	0%	25-35%
InceptionV3	90.3%	31.5%	55-65%
Decision Tree (1,2-gram)	99.4%	61.5%	70-90%
Naïve Bayes (1,2-gram)	70.1%	53.7%	40-60%
SVM (1,2-gram)	99.2%	74%	70-90%
Decision Tree (gadget)	99%	99%	99%
Naïve Bayes (gadget)	98%	98%	98%
Naïve Bayes (gadget)	99%	99%	99%

Table 18. Results for Targeted attacks.

	1-layer	*2-layer*	*3-layer*	*4-layer*	*5-layer*
Flatten	0	0	0	0	0
Virtualize	0	0	0	0	0
Encode L.	0	0	0	0	0
Encode A.	0	0	0	0	0
Opaque	0	0	0	0	0

Outside of the CNNs, the generated samples were shown to have an effect on the opcode-based models but not to the same degree. We can assume the reason for this is

that these samples were created from our CNNs which, while they do take into account opcodes through the pixel values, are also concerned with the spatial aspects of the image itself. Our opcode classifiers would most likely be more effected by methods that focus on increasing counts or other derived metrics.

Our adversarial examples had the least effect on the gadget-based classifiers. This is due to the inserted code being semantic nops. Since the code is functionally "dead" it was unable to make any changes to the gadget list of a sample. While this means that the accuracy of these classifiers is unaffected, we do not view this as a failure as increasing the gadgets present in a program is by no means an improvement to a program's security.

6.1.6 Future Work

The future work for this research, involves broadening the scope of the attack outside of classifiers that rely on image analysis. As seen in our results, models that do not take any information specific to image analysis will only have minor impacts and even then, only after large changes. We believe that this attack can be expanded to include means of attacking those types of models as well, either through larger, more involved code insertion or by adding additional methods of modification to the process.

6.1.7 Conclusion

In this section, we have proposed and evaluated an extension to existed methods for creating functional adversarial examples for from software binaries. This modified algorithm, which we call Adversarial Obfuscation, was capable of using distance measurements and semantic nop insertion in order to construct adversarial examples from obfuscated programs. These adversarial examples were shown to be effective at reducing the classification accuracy of CNN that rely of code image analysis. We showcased the

effectiveness of this algorithm in both a white and black box setting, with our samples able to reduce the accuracy of our models under attack to as low as 0%. Even the more robust InceptionV3 model was brought below 50% in a black box setting.

We further showed that our approach is effective on layered obfuscations as well, with our models misclassifying samples with 2 or more transformations present. This approach was extended to successfully perform targeted attacks as well, both for single and layered transformations.

6.2 Opcode Expansion

This section introduces and analyzes our opcode expansion approach to defeating automated code analysis. This approach involves algorithmically adding opcodes to a program in order to modify the feature set of the program. This can be viewed as a specific variant of adversarial example crafting.

6.2.1 Introduction

This research introduces and explores the process of opcode expansion. A method of modifying the opcode composition of an executable in order to deter metadata recovery attacks using machine learning. As shown in the previous section, traditional adversarial example creation methods based on image analysis, may not be as effective when transferred to other supervised learners that make use of different feature sets, such a pure opcode n-gram. Much like the methods shown in the previous section, the goal of this approach is to modify the samples via guided dead code insertion, to produce a semantically equivalent program that has improved stealth against machine learning.

The core of opcode expansion is to selectively add segments of dummy code into a program in order shift the feature sets derived from opcodes, either in a given direction for a targeted attack or just away from the original in an untargeted attack. We explore three approaches for expansion: Uniform, Profile, and Target. We evaluate the effectiveness of this method by attempting to classify the generated samples with our suite of supervised learning classifiers. Two of our methods are black box attacks, while one is white box but can performed in the same style as a substitution attack. As before, we test this method on both single and multi-layered obfuscations.

We make the following contributions:

- We evaluate the approach of Opcode Expansion, a method for generating adversarial obfuscated binaries and for algorithmic dead code insertion.
- We show that our expanded binaries are effective in black box attacks via adversarial transference.
- We explore the effectiveness our method against classifiers that make use of image analysis.
- We compare samples generated using our method against the approaches of instruction substitution and random dead code insertion.

6.2.2 Methodology

The methodology for this research was introduced in Chapter IV, and we elaborate on it further in this section. The goal of our expansions is to alter the count, or other derived statistic, of opcodes within a program to predetermined levels to decrease the effectiveness of automated metadata recovery. To expand the code, we insert dead code segments of varying lengths. We first obtain the assembly of our obfuscated c

source files in the same manner as Chapter V. For our uniform expansion, we insert code to ensure that the metric in question for all opcodes are equal. For profile expansion, we must first profile the various obfuscating transformations and then use the profiles to alter the opcodes of our files. Our last expansion type is guided expansion, and it will be detailed in a later section.

For our expansion, we expand both specific opcodes and opcode groups. Opcodes in x86 are divided into four groups:

- Arithmetic: the mathematic instructions such as add, sub, and multi.
- Data: the instructions that deal with processing of data such as mov, load, pop, and push.
- Control: instructions that handle the control flow of a program such as jmp, call, and ret. The nop instruction is considered a control instruction.
- Logic: instructions that act as checks/gates such as and, or, and test. These are often paired with control instructions.

Instead of modifying specific opcodes, we can make use of these groups for our expansions. This can give us the ability to modify the statistics of opcodes that are rarely used in dead code samples and allow for more variety in expansion. An outline of our methodology can be seen in Figure 33.

6.2.3.1 Code Segments. Before performing counts, we first determine the number of segments we wish to divide the code into. Code can be divided into a number of segments 1>= s <= total length, with s being the number of segments and total length being the total number of instructions within the assembly. The purpose of these segments is to localize and better target the effects of our expansions. Instruction counts

and code insertion will be on the divided sections. We segment our samples by the powers of 2 but any number or system could be used so long as it falls within the parameters. Code could also be segmented via program structures such as functions, logic, or basic blocks.

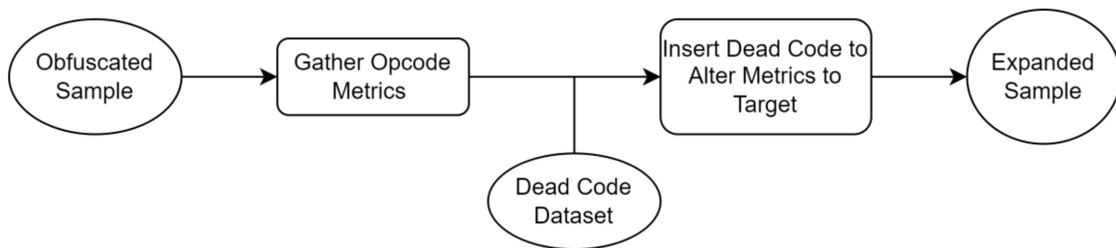

Figure 33. Outline of expansion process. Metric targets are based on the expansion type.

6.2.3.2 Uniform Expansion. For uniform expansion, the goal is to ensure that, for a given metric, every opcode or group is equal within the program segments. For this work, we choose to expand based on counts. This can be done in one of three ways, equaling every count to the highest single count across the segments, finding the highest counts per instruction/group and equaling those, or setting a target higher than any given count. For example:

- Per option 1: If a sample is dived into 16 segments and the count for control is 42 in one segment, then all groups would be expanded to 42 in each segment.
- Per option 2: If within the same sample the highest counts for the groups across all segments is 42, 27, 18, and 12, then every segment will be expanded to 42, 27, 18, and 12.

- Per option 3: Based on the highest count found, 42, a target would be chosen above that, such as 64. All counts would be increased to this number. For our purposes, we use again use powers of 2.

Regardless of the option chosen, after identifying our target counts, we limit increases to this count to avoid unnecessary expansions. The code is then processed segment by segment, inserting varying lengths of dead code to reach the target count. We insert longer code segments first to have a greater impact on the counts, then switch to progressively smaller segments as more precise changes are needed.

6.2.3.3 Profile Expansion. For profile expansion, instead of expanding to achieve uniformity within the program, we expand the opcodes to match a profile, a predetermined set of opcode metrics taken from another program. For this research, we are expanding to obfuscation profiles. We first form these profiles before beginning expansions. Similar to uniform expansion, we continue to work with opcode count.

Creating obfuscation profiles is the first step. We create two different types of obfuscation profiles, average and instance. To form our average profiles, we first take our obfuscated dataset and perform our opcode counts on all samples containing the a given transformation. These counts are then averaged together to form a general profile for the given obfuscation. In addition to the counts, we also take the average counts for our clean samples and measure the percentage change between the clean and obfuscated profile. These counts are also taken at different segmentation levels as well. It is important to note, an average profile is only formed from samples that contain only the given transformation. Samples with layered transformations are not included in the averages but could be used to form averages for various combinations. We do not explore this.

For our instance profiles, we first take the sample to be expanded and obfuscate this sample with one or more new transformations. This new variant is then segmented and counted to form a profile. We measure the percentage change between this new variant and the original as well. The profile now represents a given transformation specific to one instance.

After profiles are formed, we expand the target in much the same way as with uniform expansion. The key differences are that segments are now expanded to the matching segment from the profile. In addition, segments can also be expanded to the percentage targets, rather than just to the raw counts.

6.2.3.4 Guided Expansion. The last expansion approach we explore is targeted expansion. This expansion can be viewed as an adversarial machine learning attack as it relies on exploiting information gained from a machine learning model. For our approach, we rely on the decision tree model created in previous chapters. Decision trees are unique as it is considerably easy to view and understand why the model makes the decisions that it does. We can exploit this to expand a sample in certain ways to force a different decision.

This attack can happen in one of two ways, the first relies specifically on decision trees and the second just relies on understanding the feature importance of the model. For the method, we analyze the generated decision to tree to understand the decisions that lead to the file being classified correctly (or incorrectly in a target attack). We can then modify the opcode metrics of the sample to push it down a different branch of the tree. Since this attack is dependent on the type of features the tree is using, this process may be more involved than simply matching counts. In our work, the decision tree uses TF-IDF

as its feature value, so the TF-IDF of opcodes must be calculated and used to guide the opcode insertion.

The second approach can be used both with a decision tree and also for other models. First a model is created to accurately classify samples. This model can then be analyzed to find the features that are considered most impactful to the decision-making process. These features can be compared to another sample that is classified differently or modified at random in order to push the classification in a different direction. We explore both of these approaches. Table 19 shows the total number of features for each transformation, as well as the top 50 features. These are the features that would be targeted during guided expansion.

6.2.3 Results

The following showcase the results from the various tests performed using opcode expansion. These results will be discussed in the next section.

6.2.3.1 Opcode Profiles. Table 20 shows the average opcode group counts and percentage increases from the various transformations that we considered. For this purpose the *flatten* and *bogus control-flow* transformation from OLLVM are considered under the *Flat* and *Opaque* groups together with the equivalent Tigress transforms.

6.2.3.2 Uniform. For our uniform expansion test, we selected 1,000 files at random and expanded the files to uniformity based on the three metrics for uniformity presented above. After expansion, these files were presented to our classifier suite. Table 21 shows the results of options 1,2, and 3 with our classifiers at being trained on multi-layer sample.

Table 19. Feature amount and top 50 features.

Encode A:
Selected features: 213
addl addl,addl andl,addl orl,addq,addq movb,andl,andl addl,andl leal,andl movl,andl sarl,andl subl,cmpb,cmpb jne,cmpl movzbl,cmpq,cmpq jne,jg leaq,jmp,jmp cmpb,jne,jne movq,jns,jns leaq,jns movl,js,js leaq,js movl,leal,leal movl,leal movzbl,leal movzwl,movabsq addq,movb addl,movl,movl addl,movl movl,movl movslq,movl orl,movl subl,movl subq,movl testl,movl xorl,movq,movq addq,movq cmpq,movq jmp,movq movq,movq orq,movslq,movslq movq
Encode L:
Selected features: 295
addl,addl jmp,addl movl,addl nop,addq,addq movb,addq movq,andl,call,call call,cmpb,cmpb jne,cmpl movzbl,cmpq,cmpq je,cmpq jne,je cmpl,jg cmpl,jmp,jmp cmpb,jmp movq,jne,jne movq,leaq,leaq addq,movb,movb addl,movl movl,movl movq,movl movslq,movl subl,movq,movq addq,movq cmpq,movq jmp,movq movq,movq subq,movslq,movslq movq,movzbl,movzbl andl,movzwl,nop nop,subl,subl cmpl,subl movl,subq,jmp cmpl,movq leaq,movq movl
Flat:
Selected features: 229
addl movl,addl movq,addq movb,addq notrack,call leaq,call movq,cmpb jne,cmpl jg,cmpl movzbl,cmpq ja,ja,ja movq,je jmp,jg,jg movq,jmp,jmp call,jmp cmpb,jmp cmpl,jmp jmp,jmp movl,jmp movq,leaq,movb addl,movl movq,movl movslq,movl subl,movq addq,movq jmp,movslq,movslq movq,notrack,notrack jmp,subl,subl cmpl,cmpb,cmpq je,cmpl jle,jle,addq movq,subl movl,addl cmpl,movq movq,cmpq jne,jne,andl sarl,leaq leaq,movl jmp,jmp movzbl,leaq movl
Opaque:
Selected features: 204
addl,addl movl,addq,addq movb,addq movq,call jmp,call movq,cmpb,cmpb jne,cmpl movzbl,cmpq,cmpq je,cmpq jne,je movq,jle movq,jmp cmpb,jmp jmp,jne,jne cmpl,jne jmp,jne movl,leaq subq,movb,movb addl,movl movslq,movl subl,movq,movq addq,movq cmpq,movq jmp,movq movq,movslq,movslq movq,movzbl andl,subl,subl cmpl,subl movl,subq leaq,jmp cmpl,movl call,movq movl,jne movq,andl,je,sarl,movq leaq,movzbl,jne leaq,andl sarl,movl xorl
Virtualize:
Selected features: 300
addl,addl movl,addq,addq movb,addq movl,addq movq,addq popq,andl,call,call addq,call jmp,call movl,cmpb,cmpb jne,cmpl ja,cmpl movq,cmpl movzbl,cmpq,cmpq je,cmpq jne,ja movl,jg testl,jmp cmpb,jmp movl,jne,jne movq,leaq movl,movb,movb addl,movb cmpb,movl,movl call,movl cmpl,movl movl,movl movslq,movl subl,movq,movq addq,movq cmpq,movq jmp,movq leaq,movq movq,movq movslq,movq pushq,movq subq,movslq,movzbl,movzbl andl,movzbl movzbl,movzwl

Table 20. Average instruction group counts and percentage increase.

	Average Counts				*Average Percentage Increase*			
	Arith.	Logic	Data	Control	Arith.	Logic	Data	Control
Virtualize	124.67	43.96	316.33	87.60	79.17	69.67	79.87	76.79
Flatten	33.6057	16.21	102.77	42.11	11.88	6.21	29.32	44.93
Encode A	46.94	32.02	101.87	25.57	44.70	58.36	37.49	20.50
Encode L	117.59	16.80	269.01	43.85	77.92	20.64	76.33	53.64
Opaque	29.49	27.47	164.75	55.49	12.00	51.47	61.35	63.37

6.2.3.3. Profile. Our profile-based expansion test utilizes another 1,000 files chosen once again at random. To perform this test, the profile counts and averages were used to move the samples into an obfuscation not currently present in the sample. This means that no 5-layers samples were able to be used. Table 22 shows the results of profiles formed with counts created from instance profiles, as well as the accuracies formed from average percentage profiles.

Table 21. Classification results for uniform expansion.

	Original	*Option1*	*Option 2*	*Option 3*
CNN 1	90%	76%	84%	53%
CNN 2	89%	72%	87%	54%
CNN 3	89%	71%	85%	53%
InceptionV3	90%	83%	89%	64%
Decision Tree (1,2-gram)	99%	88%	78%	46%
Naïve Bayes (1,2-gram)	70%	62%	52%	14%
SVM (1,2-gram)	99%	87%	81%	38%
Decision Tree (gadget)	99%	95%	91%	94%
Naïve Bayes (gadget)	98%	93%	92%	83%
SVM (gadget)	99%	94%	94%	92%

Table 22. Classification results for profile expansion.

	Original	*Instance*	*Average*
CNN 1	91%	89%	84%
CNN 2	88%	87%	87%
CNN 3	90%	87%	88%
InceptionV3	92%	89%	90%
Decision Tree (1,2-gram)	98%	95%	94%
Naïve Bayes (1,2-gram)	65%	61%	63%
SVM (1,2-gram)	97%	93%	92%
Decision Tree (gadget)	99%	98%	97%
Naïve Bayes (gadget)	98%	96%	96%
SVM (gadget)	99%	98%	97%

<u>6.2.3.4 Guided.</u> Guided expansion was tested using a smaller sample set for the decision tree-based expansion. This is due to parts of this process not yet being fully automated. Table 23 showcases the accuracies of our tree models for files expanded both targeted and non-targeted. All models used in this test were trained on multi-layer samples.

Table 23. Classification results for guided expansion.

	Untargeted			*Targeted*		
Trees	1-gram	1,2-gram	1,3-gram	1-gram	1,2-gram	1,3-gram
Encode A	5%	0%	15%	10%	0%	15%
Encode L	10%	0%	5%	5%	0%	10%
Virtualize	5%	0%	0%	10%	0%	20%
Flatten	0%	0%	5%	5%	0%	15%
Opaque	10%	0%	10%	15%	0%	0%

Table 24. Guided expansion classification results.

	Original	*Guided Expansion*
CNN 1	90.8%	22.5%
CNN 2	89.4%	21.7%
CNN 3	89.7%	22.8%
InceptionV3	92.1%	43.2%
Decision Tree (1,2-gram)	99.7%	14%
Naïve Bayes (1,2-gram)	71.2%	0%
SVM (1,2-gram)	99.4%	27%
Decision Tree (gadget)	99%	92%
Naïve Bayes (gadget)	98%	88%
SVM (gadget)	99%	91%

The next test for Guided Expansion utilized the highest performing features for our 2-gram decision tree. For the untargeted attack, we inserted to dead code with the intention of lowering or raising the frequency of opcodes that were identified as valuable features. For the targeted attack, we compared samples of the target transformation to our samples and adjusted the frequency of the features accordingly. Table 24 showcases the results of our models attempting to classify samples that were expanded based on this extracted feature importance.

6.2.4 Discussion

Looking over the results, there are few key takeaways and points worth exploring, starting with profile expansion. The opcode group profiles provide a good look at the impact of the obfuscating transformations on the opcode makeup of a binary. This information can be useful to heuristic approaches to code analysis both inside and out of

our work. Unfortunately, expanding a programs opcode profile based on these did not have the desired result. Neither instance nor average based expansion were able to achieve significant impact to the performance of our supervised learners. We believe that this method still holds promise and that it can be further refined.

Our next tests, had to deal with uniform based expansion. We explored all three of the presented options and found all to be effective and obscuring a programs metadata. Option 3 is clearly the best performance, reducing the decision tree and svm classifiers to a score below 50%. While this method was successful, it will need to be further analyzed to compare the achieved results with the impact on program cost. Especially if applied to larger programs.

Guided was the last type of expansion tested and, unlike the other methods, is at its core closer to an adversarial machine learning attack. Our first result showed that decision trees can be exploited by directly analyzing the tree structure and inserting code sequences base on the tree structure. This was even shown to work on similar trees, which would fall into the principle of adversarial transference. Our second result showed that expanding based on features extracted from a ML model can greatly enhance the impact of the expansion.

6.2.5 Future Work

Our future work for Opcode Expansion will be to continue to streamline the process and more easily automate the Guided Expansion approach. We wish to explore performing the expansions based on metrics beyond counts and for feature outside of n-gram analysis.

6.2.6 Conclusion

In this section, we have proposed and evaluated a method of algorithmic dead code insertion and a means of adversarial example generation. This approach, which we call Opcode Expansion, was capable of using opcode metrics such as counts and extracted TF-IDF values to guide dead code insertion in order to modify and improve the stealth of obfuscated programs. These expanded examples were shown to be effective at reducing the classification accuracy of supervised learners that real on opcode features. We showcased the effectiveness of this algorithm in both a white and black box setting, with our samples able to reduce the accuracy of our models under attack to as low as 0%. Our model was also shown to be effective at reducing the accuracy of image based CNNs.

We further showed that our approach is effective on layered obfuscations as well, with our models misclassifying samples with 2 or more transformations present. This approach was extended to successfully perform targeted attacks as well, both for single and layered transformations.

CHAPTER VII

IMPLEMENTATION AND EVALUATION

This chapter details our proposal of an obfuscation framework that takes advantage of methods discussed in previous chapters to improve the overall stealth of obfuscations. We measure the metrics of code produced by the proposed framework to show that it has no negative impacts to obfuscating transformations.

7.1 Introduction

In recent chapters, we detailed the capabilities of supervised machine learning to analyze and detail both obfuscating transformations using different feature sets, as well as some fine-grained components of the transformation. We then described two methods of that can be used to modify obfuscate binaries in order to improve the stealth of those binaries against automated analysis. As both of these methods has successes and shortcomings, we propose the creation of a unified method that combines the best of both algorithms. We dub this approach Adversarial Expansion. At the simplest, it is the previous two methods applied at the same time to a sample, ensuring that both maintain the full stealth benefit. We propose and showcase a way to truly combine the two approaches, that allows the expansions from Opcode Expansion to be guided by the distance analysis of Adversarial Obfuscation.

To accomplish this, we propose the LOKI Obfuscation framework. Loki will be a collection of scripts and processes that will rely on other tools to enhance the obfuscation of files from beginning to end. For this research, we only showcase LOKI's ability to improve the stealth of a file via Adversarial Expansion and leave further showcases to future work. We evaluate the Adversarial Expansion process against the same suite of classifiers to examine the stealth impacts. We also use cyclomatic complexity and analysis with GCC to test that LOKI's changes have not negatively impacted potency or resilience.

- A unified approach to improving the stealth of obfuscation via Adversarial Expansion.
- The proposal of the LOKI framework in order to fully automate the process and guide the obfuscation process
- We examine the metric impact of Adversarial Obfuscation to ensure that potency and resilience are maintained.

7.2 LOKI Obfuscator Framework

We believe that machine learning can assist in the obfuscation of programs in many ways. To that end and to make full use of the information gained in this research, we propose the creation of the LOKI framework. LOKI exists as a collection of scripts and modules created in Python that will link to various tools in order to guide the obfuscation process. LOKI will make use of various machine learning obfuscation research outside of the adversarial research shown in this work. LOKI will be open -

source and available of Github once it more pieces of the framework are complete. A brief overview of LOKI's proposed capabilities is described in the following sections.

7.2.1 Capabilities

As is currently planned, LOKI will consist of modules, which themselves are collections of scripts. LOKI will have modules for each of the capabilities described in the following sections.

7.2.1.1 Guided Obfuscation. The guided obfuscation module will be the module used to begin the obfuscation process. The capabilities of this module will be to modify the initial obfuscation process in order to achieve high results in potency and resilience for the source code and chosen obfuscations. Examples of this include proposing the strongest ordering of layered obfuscations, the depth of arithmetic encoding, and the choice of options for a variety of other transformations. Some machine learning research will benefit his module, such as neural networks that can predict potency and resilience.

7.2.1.2 Adversarial. The Adversarial module will be how LOKI implements the three adversarial processes described in this work. There will be script collections for Adversarial Obfuscation, Opcode Expansion, and Adversarial Expansion. The obfuscation and expansion scripts will drive the processes detailed in Chapter VI. These scripts will support the implementation of the two basic types of expansion (uniform and profile). Guided expansion will be added to the framework once the process becomes more easily automated.

Adversarial Expansion is detailed further in this chapter will be implemented shortly after the initial deployment of LOKI.

7.2.1.3 Obfuscation Analysis. Analysis will be where scripts relating to the analysis of obfuscations will be kept. This analysis will primarily consist of using machine learning and other tools to analyze metrics and features present in obfuscated files.

7.3 Methodology

This section describes the methodology used in this chapter. We describe the process of Adversarial Expansion, which is a combination of the previous two algorithms described in this work. We will perform adversarial expansion and then test the impact on the four obfuscation metrics. The goal of this algorithm is only to improve stealth and avoid damaging any of the other metrics. For testing the metrics:

- Stealth will be analyzed using the misclassification rate of classifiers,
- Potency will be measured using the cyclomatic complexity of the samples,
- Resilience will be measured by compiling the code with various optimization levels and seeing the impact to code size,
- Cost will be measured by examining code size and runtime.

7.3.1 Adversarial Expansion

Adversarial expansion is the combination of adversarial obfuscation and opcode expansion. This will be implemented in two ways. The first will be to simply apply the two algorithms on top of each other. Both algorithms will be in a closed loop to ensure that the goals of both algorithms are met, without one damaging the other.

The second method involves having the methods feed into each other. First, guided obfuscation will form of a list of dead code snippets, L, that must be inserted to

evade the n-gram feature analysis. After this list is given, it will replace the list of semantic nops typically used by adversarial obfuscation. An image will be generated using FGSM and CW. The binary distance metric will be used to reduce the distance between the code binary and the non-functioning AE. Insertion will proceed as normal, except with the standard list being replaced by the list from expansion. Some of the semantic nops may be present in the new list but it will also contain the longer samples used in expansion. The overall process can be seen in Figure 34. This method will implement the waterfall method as well for multiple labels and targeted attacks.

If Adversarial Obfuscation is unable to achieve the desired misclassification rate, nop instructions can be inserted throughout the program. The locations of these instructions will replace the traditional insertion points and the algorithm will resort to using the standard semantic nop list, similar to [109].

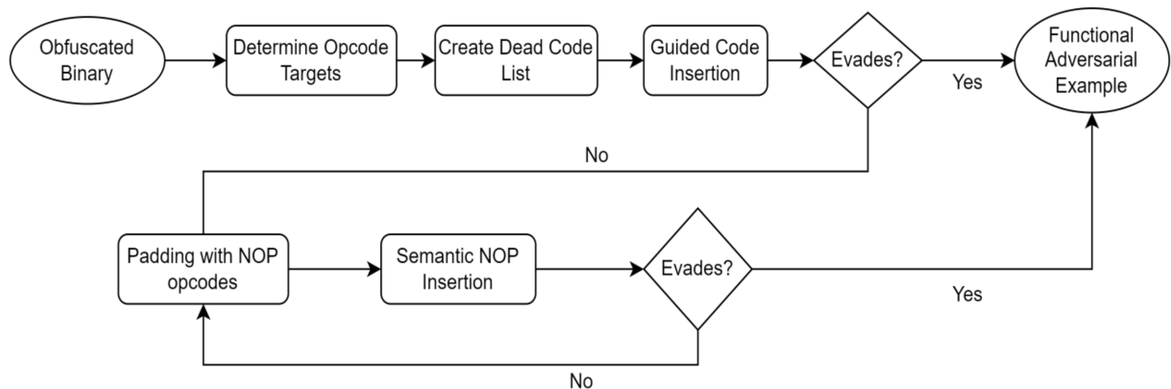

Figure 34. Adversarial Expansion Process.

7.3.2 Metrics

After adversarial expansion has completed, the samples will be analyzed to determine the impact that adversarial expansion has on the obfuscation metrics. The metrics themselves are defined in Chapter II. The goal is to decrease stealth, while having no negative impact on potency and resilience. For cost, the goal is to minimize the impact as it is unavoidable.

7.3.2.1 Stealth. Stealth is a measure of how well obfuscation blends in with the surrounding code. As there is no accepted measurement for stealth, we choose to define stealth as the classification accuracy of machine learning algorithms. Our samples will be fed through our suite of supervised learning to determine the classification accuracy of the models. Both the original and expanded samples will be given to the models to compare the results.

7.3.2.2 Potency. Potency is a measure of how dissimilar and complicated obfuscation has made the original code. For this metric, we will measure the cyclomatic complexity (CC) of the code. CC is used to measure the stability and level of confidence in a program [110]. Programs with lower CC are considered easier to understand and modify. As such, higher CC can be used to determine the impact of an obfuscation. CC uses the control flow graph and can be calculated with the following formula: $M = E - N + 2P$. Where E is the number of edges in the graph, N is the number of nodes, and P is the number of connected components.

7.3.2.3 Resilience. Resilience is described as the difficulty in removing the obfuscations on a program. We are using GCC compiler optimization options (0, 1, 2, 3, s, fast) to measure the amount of time or resources taken to deobfuscate the transformations with

and without Adversarial Expansion. We will use file size as the primary metric for determining the resilience. Higher size after the optimizations implies more resilient changes.

7.3.2.4 Cost. Cost will be measured as the file size in bytes of the samples at optimization level 0 with GCC.

7.4 Results

The following sections detail the results of our methodology.

7.4.1 Impact on Stealth

We continue to define stealth as the classification rate of our machine learning algorithms. We run our samples and the originals through the multi-layer variants trained in Chapter V. For the CNN the presented score is the average of the individual layered networks. We do not make use of the FCNN shown in Chapter V. Table 25 shows the classification scores.

7.4.2 Impact on Potency

Table 26 shows the results of using the CC measurement on the original obfuscated binaries and on the same binaries modified via Adversarial Expansion. As we can see by examining the table, the dummy code inserted into the file has increased the cyclomatic complexity of the samples. No sample was negatively impacted, and we can observe that the impact of the dummy code is dependent on the type of transformation.

Table 25. Classification scores for our generated samples.

	ORIGINAL	ADVERSARIAL EXPANSION
CNN 1	90.8%	0%
CNN 2	89.4%	0%
INCEPTIONV3	90.3%	20.7%
DECISION TREE (1-GRAM)	98.15	22.7%
DECISION TREE (1,2-GRAM)	99.4%	23.4%
DECISION TREE (1,2,3-GRAM)	97.2%	22.6%
NAÏVE BAYES (1-GRAM)	65.3%	0%
NAÏVE BAYES (1,2-GRAM)	70.1%	0%
NAÏVE BAYES (1,2,3-GRAM)	72.%	0%
SVM (1-GRAM)	99.5%	24.1%
SVM (1,2-GRAM)	99.2%	27.7%
SVM (1,2,3-GRAM)	99.4%	21.6%

Table 26. Cyclomatic complexity measures for transformation combinations.

	Original	*Adversarial Expansion*
A	1.7	6
ALO	5.4	10.1
AO	8.0	14.7
F	3.7	5.2
FAL	2.8	6.7
FO	9.0	16.9
FOAL	6.2	20.3
L	1.6	5
O	7.3	11.5
OA	7.3	12.3
OAL	5.0	12.1
OF	22.3	24.9
OL	4.9	8.2
V	12.3	16.0
VAL	9.5	13.3
VF	34.7	41.2
VFOAL	57.0	71.3
VO	22.7	29.1
VOAL	62.0	78.1
VOLAF	80.5	93.4

7.4.3 Impact on Resilience

Table 27 shows the results of our evaluation of the samples resilience. As stated previously, we made use of the gcc compiler to compile the code with varying levels of optimizations. As optimization is the inverse of obfuscation, the serves as a stand in for an attacker performing deobfuscation. We can observe that for each sample, our expanded variants were able to maintain more of their code through the optimization process. Our samples were even shown to have varying degrees of resistance against differing obfuscation levels, unlike the base samples.

Table 27. Sample Size Comparison.

	ORIGINAL					ADVERSARIAL EXPANSION				
	0	1	2	3	s	0	1	2	3	s
A	55033	54521	54629	54629	54629	57024	56000	56620	56795	56108
ALO	57016	55992	56100	56100	56100	62034	58898	59021	59102	58921
AO	56412	55388	55496	55496	55496	59904	57423	57598	57609	57481
F	55545	54521	54629	54629	54629	57541	56953	57014	57134	56988
FAL	55673	55125	55745	55745	55233	58300	57372	57598	57629	57419
FO	56412	55900	55496	55496	55496	60849	58492	59837	59975	58637
FOAL	57564	56028	56100	56100	56100	65007	61734	62938	62987	62029
L	55637	55125	55233	55233	55233	57403	55394	55419	55498	55419
O	56412	55388	55496	55496	55496	59513	57852	57902	57937	57871
OA	56924	55388	55496	55496	55496	59637	58879	58930	58965	58903
OAL	58730	56170	56278	56278	56278	62348	59748	59948	60027	59839
OF	57436	56412	57032	57032	56520	60594	57858	58963	59074	58756
OL	58218	56170	56278	56278	56278	61120	58642	58673	59003	58679
V	56645	56133	56241	56241	55729	61123	60178	60258	60097	60204
VAL	58220	56172	56792	56792	56280	62637	58938	59103	59340	58985
VF	58693	57157	57777	57777	56753	63902	60386	60847	61084	60743
VFOAL	73423	62671	62267	62267	63291	82672	67891	68122	68496	68038
VO	59048	56488	57108	57108	56596	65657	59645	60464	60931	60004
VOAL	85711	57551	57659	57659	57659	93976	78348	79436	79614	78509
VOLAF	90319	64719	64827	64827	64315	101,661	68286	69334	69579	68589

7.4.4 Impact on Cost

Referring again to Table 27, we can see that the size of the samples did increase by an amount roughly >= 10,000 bytes for each sample. While this is unfortunate, we believe that this can be mitigated with a more efficient and varied dead code database. We can also observe that the increase in size due not appear to scale at a large rate with file size.

7.5 Discussion

Reviewing the results for the metric testing of adversarial expansion, we are confident in saying that the algorithm achieves its desired results. In the stealth test, the samples were able to achieve a misclassification rate of below 50% for all classifiers with the exception of the gadget-based classifiers. InceptionV3 is the only CNN model to be above 0% with a score of 20.7%, and the n-gram based decision trees and SVMs have been reduced to scores in range of 20% - 35%. This is a vast improvement over the two individual methods, which were unable to seriously impact the type of classifier that they were not constructed for. This makes Adversarial Expansion as our strongest algorithm for generating adversarial examples.

7.6 Future Work & Conclusion

In this section, we proposed and showcased the combined version of our two prior adversarial creation methods (adversarial obfuscation and opcode expansion) Adversarial Expansion. This method uses the dead code segment selected as part of guided expansion to form the insertion list for a round of adversarial obfuscation. Our samples generated

with Adversarial Expansion, were shown to have the benefits of both methods and are capable of evading both the image based neural networks and the N-gram based opcode classifiers. With this method, several of the limitations encountered previously are overcome.

After testing the metric impacts of Adversarial Expansion on our obfuscated samples, we found that all goals were achieved across the four metrics. For stealth, the classification accuracy was reduced to as low as 0%, with both categories of classifiers being affected. This was done without requiring the heavy-handed approach of uniform expansion. Potency was not only maintained across the range samples, but it was also as much as doubled for the weaker transformation. A similar situation was observed with resilience. As the initial goal was simply to avoid a negative impact, this means the method performed above expectation. The final metric of cost was impacted negatively; however, the impact was not extreme and could be mitigated with a larger dead code base to pull from and a more mature version of the algorithm.

For future work, the first and main goal is to further automate the adversarial expansion process. Much of the work is done by hand with automation only assisting in certain parts of the process. A fully automated process will allow for further improvements to be made and for the process to be broadened more easily to other feature sets and applications. In addition, once the process has been hard coded and automated, the scripts can be compiled into modules and the LOKI framework can be made live.

REFERENCES

[1] "Prelude: Computing from Ancient Times to the Modern Era," *InformIT*, 2013. [Online]. Available: http://www.informit.com/articles/article.aspx?p=2163344&seqNum=5 [Accessed February 13th, 2021].

[2] "The Growing $1 Trillion Economic Impact of Software," *BSA Foundation*, 2017. [Online]. Available: https://software.org/reports/2017-us-software-impact/ [Accessed March 3rd, 2021].

[3] "The Cost of Malicious Cyber Activity to the U.S. Economy," The Council of Economic Advisers - Executive Office of the President of the United States, Feb. 2018. [Online]. Available: https://www.whitehouse.gov/wp-content/uploads/2018/02/The-Cost-of-Malicious-Cyber-Activity-to-the-U.S.-Economy.pdf.

[4] "Intellectual Property Rights in Software – What They Are and How to Protect Them," *Freibrun Law*. [Online]. Available: https://freibrun.com/intellectual-property-rights-software-protect/ [Accessed March 3rd, 2021].

[5] J. Nagra and C. Collberg, *Surreptitious software: obfuscation, watermarking, and tamperproofing for software protection*. Pearson Education. 2009.

[6] C. Collberg, C. Thomborson, and D. Low, “A taxonomy of obfuscating transformations,” Department of Computer Science, The University of Auckland, New Zealand, Tech. Report. 1997.

[7] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation - tools for software protection," *IEEE Transactions on Software Engineering*, vol. 28, no. 8, pp. 735-746, Aug. 2002, doi: 10.1109/TSE.2002.1027797.

[8] P. Falcarin, C. Collberg, M. Atallah, & M. Jakubowski, “Software Protection.” [Online]. Available: https://www2.cs.arizona.edu/people/collberg/content/research/papers/falcarin11software.pdf [Accessed March 3rd, 2021].

[9] Security, P., 2021. What is Software Piracy? - Panda Security Mediacenter. [online] Panda Security Mediacenter. Available at: <https://www.pandasecurity.com/en/mediacenter/panda-security/software-piracy/#:~:text=According%20to%20the%202018%20Global,know%20about%20the%20software%20laws.> [Accessed 27 March 2021].

[10] “BSA Global Software Survey: Seizing Opportunity Through License Compliance,” *Bsa.org*, 2016. [Online]. Available: https://globalstudy.bsa.org/2016/ [Accessed: March 26, 2021].

[11] T. Miracco, “The Hidden Cost of Software Piracy in The Manufacturing Industry,” *Manufacturing.net*, August 12, 2018. [Online]. Available: https://www.manufacturing.net/article/2016/02/hidden-cost-software-piracy-manufacturing-industry [Accessed March 3rd, 2021].

[12] J. M. Memon, A. Khan, A. Baig, & A. Shah, “A study of software protection techniques,” in *Innovations and Advanced Techniques in Computer and Information Sciences and Engineering*. Dordrecht: Springer, 2007, pp-249-253.

[13] S. K. Udupa, S. K. Debray, and M. Madou. Deobfuscation: Reverse engineering obfuscated code. In Reverse Engineering, 12th Working Conference on, pages 10–pp. IEEE, 2005.

[14] A. Salem and S. Banescu, “Metadata recovery from obfuscated programs using machine learning,” Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering - SSPREW 16, 2016

[15] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray. A generic approach to automatic deobfuscation of executable code. Technical report, technical report, Department of Computer Science, The University of Arizona, 2014.

[16] A. Slowinska, T. Stancescu, and H. Bos. Howard: A dynamic excavator for reverse engineering data structures. In NDSS. Citeseer, 2011.

[17] L. Jones, D. Christman, S. Banescu and M. Carlisle, "ByteWise: A case study in neural network obfuscation identification," 2018 IEEE 8th Annual Computing

and Communication Workshop and Conference (CCWC), Las Vegas, NV, USA, 2018, pp. 155-164, doi: 10.1109/CCWC.2018.8301720.

[18] Ramtine Tofighi-Shirazi, Irina Măriuca Asăvoae, and Philippe Elbaz-Vincent. 2019. Fine-grained static detection of obfuscation transforms using ensemble-learning and semantic reasoning. In Proceedings of the 9th Workshop on Software Security, Protection, and Reverse Engineering (SSPREW9 '19). Association for Computing Machinery, New York, NY, USA, Article 4, 1–12. DOI:https://doi.org/10.1145/3371307.3371313

[19] Tabassi, E., K. Burns, M. Hadjimichael, A. Molina-Markham, J. Sexton, "A Taxonomy and Terminology of Adversarial Machine Learning", NIST Technical Draft, Oct 2019, https://doi.org/10.6028/NIST.IR.8269-draft

[20] R. Manikyam, J. T. McDonald, W. R. Mahoney, T. R. Andel, & S. H. Russ, "Comparing the effectiveness of commercial obfuscators against MATE attacks," In Proceedings of the 6th Workshop on Software Security, Protection, and Reverse Engineering, December 2016, pp. 8.

[21] F. B. Cohen, "Operating system protection through program evolution," *Computers & Security*, vol. 12, no. 6, pp. 565-584. 1993.

[22] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im) possibility of obfuscating programs," In Annual International Cryptology Conference, August 2001, pp. 1-18.

[23] VMProtect Software, *VMProtect Software Protection*. [Online]. Available: https://vmpsoft.com/ [Accessed: March 26, 2021].

[24] Oreans Technology : Software Security Defined, *Themida.* [Online]. Available: https://www.oreans.com/themida.php [Accessed: March 26, 2021].

[25] *Tigress C Diversifier/Obfuscator*. [Online]. Available: http://tigress.cs.arizona.edu/ [Accessed: March 26, 2021].

[26] Xuesong Zhang, Fengling He, and Wanli Zuo. Theory and practice of program obfuscation. INTECH Open Access Publisher, 2010.

[27] C. Collberg. Tigress: Transformations Index. University of Arizona, 2015.

[28] S. Banescu, M. Ochoa, and A. Pretschner. A framework for measuring software obfuscation resilience against automated attacks. In 2015 IEEE/ACM 1st International Workshop on Software Protection (SPRO), pages v–vi, May 2015.

[29] Rolf Rolles. Unpacking virtualization obfuscators, proceedings of the 3rd usenix conference on offensive technologies. pages 1–1, 2009.

[30] B. Anckaert, M. Jakubowski, and R. Venkatesan, “Proteus: virtualization for diversified tamper-resistance.” In Proceedings of the ACM workshop on Digital rights management, October, 2006, pp. 47-58.

[31] S. Ghosh, J. Hiser, and J. W. Davidson, “Replacement attacks against vm-protected applications,” In Acm Sigplan Notices, March 2012, Vol. 47, No. 7, pp. 203-214.

[32] W. Hu, J. Hiser, D. Williams, A. Filipi, J. W. Davidson, D. Evans, ... & J. Rowanhill, “Secure and practical defense against code-injection attacks using software dynamic translation,” In Proceedings of the 2nd international conference on Virtual execution environments, June 2006, pp. 2-12.

[33] J. Salwan, S. Bardin, & M. L. Potet, “Symbolic deobfuscation: From virtualized code back to the original,” In International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, June 2018, pp. 372-392.

[34] C. Collberg, S. Martin, J. Myers, and J. Nagra. Distributed application tamper detection via continuous software updates. In Proceedings of the 28th Annual Computer Security Applications Conference, ACSAC ’12, pages 319–328, New York, NY, USA, 2012. ACM.

[35] A. Balakrishnan and C. Schulze. Code obfuscation literature survey. CS701 Construction of Compilers, 19, 2005.

[36] R. Manikyam, “Analyzing Program Protection Using Software Based Hardware Abstraction,” Ph.D. Dissertation, University of South Alabama, 2019.

[37] Murphy, K. P. (2012) *Machine Learning: A Probabilistic Perspective*, MIT Press.

[38] Bishop, C. M. (2006) “Pattern Recognition,” *Machine Learning*.

[39] N. Papernot, "Characterizing the Limits and Defenses of Machine Learning in Adversarial Settings," Ph.D. Dissertation, Pennsylvania State University, 2018.

[40] Yakura, H., Shinozaki, S., Nishimura, R., Oyama, Y., & Sakuma, J. (2017). Malware Analysis of Imaged Binary Samples by Convolutional Neural Network with Attention Mechanism. In Proceedings of the 10th ACM Workshop on Artificial Intelligence and Security - AISec '17 (pp. 55–56). New York, New York, USA: ACM Press. https://doi.org/10.1145/3128572.3140457

[41] Sutskever, I., O. Vinyals, and Q. V. Le (2014) "Sequence to sequence learning with neural networks," in *Advances in Neural Information Processing Systems*, pp. 3104–3112.

[42] Drucker, H., D. Wu, and V. N. Vapnik (1999) "Support vector machines for spam categorization," *IEEE Transactions on Neural Networks*, 10(5), pp. 1048– 1054.

[43] Das, Sayan & Barik, Rupashri & Mukherjee, Ayush. (2020). Salary Prediction Using Regression Techniques. SSRN Electronic Journal. 10.2139/ssrn.3526707.

[44] Bala, Ravula & Surya, Kunamneni & Chandravas, Tadiparthi & J., Manikandan. (2020). A Machine learning based Advanced House Price Prediction using Logistic Regression. International Journal of Computer Applications. 176. 30-34. 10.5120/ijca2020920303.

[45] Jain, A. K., M. N. Murty, and P. J. Flynn (1999) "Data clustering: A review," *ACM Computing Surveys*, 31(3), pp. 264–323.

[46] Erhan, D., Y. Bengio, A. Courville, P.-A. Manzagol, P. Vincent, and S. Bengio (2010) “Why does unsupervised pre-training help deep learning?” *Journal of Machine Learning Research*, 11, pp. 625–660.

[47] Hu, J. and M. P. Wellman (2003) “Nash Q-learning for general-sum stochastic games,” *Journal of Machine Learning Research*, 4, pp. 1039–1069.

[48] Sutton, R. S. and A. G. Barto (1998) *Reinforcement Learning: An Introduction*, MIT Press.

[49] G. McGraw, H. Figueroa, V. Shepardson, and R. Bonett, “An architectural risk analysis of machine learning systems: Toward more secure machine learning,” *Berryville Institute of Machine Learning*, 2020. [Online]. Available: https://www.garymcgraw.com/wp-content/uploads/2020/02/BIML-ARA.pdf [Accessed on: Mar, 23].

[50] Sommer, R. and V. Paxson (2010) “Outside the closed world: On using machine learning for network intrusion detection,” in *IEEE Symposium on Security and Privacy*, pp. 305–316.

[51] Goodfellow, I., Y. Bengio, and A. Courville (2016) “Deep Learning,” Book in preparation for MIT Press (www.deeplearningbook.org).

[52] L. Huang, A. D. Joseph, B. Nelson, B. I. P. Rubinstein and J. D. Tygar, "Adversarial Machine Learning," in Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence, New York, NY, USA, 2011.

[53] Kloft, M. and P. Laskov (2010) "Online anomaly detection under adversarial impact," in *13th International Conference on Artificial Intelligence and Statistics*, pp. 405–412.

[54] Barreno, M., B. Nelson, R. Sears, A. D. Joseph, and J. D. Tygar (2006) "Can machine learning be secure?" in *ACM Symposium on Information, Computer and Communications Security*, pp. 16–25.

[55] Kloft, Marius, and Pavel Laskov. "A poisoning attack against online anomaly detection." In NIPS Workshop on Machine Learning in Adversarial Environments for Computer Security. 2007.

[56] A. Chakraborty, M. Alam, V. Dey, A. Chattopadhyay and D. Mukhopadhyay, "Adversarial Attacks and Defences: A Survey," 28 9 2018.

[57] N. Papernot, P. McDaniel, A. Sinha and M. P. Wellman, "SoK: Security and privacy in machine learning," in 2018 IEEE European Symposium on Security and Privacy (EuroS&P), 2018.

[58] Q. Liu, P. Li, W. Zhao, W. Cai, S. Yu and V. C. M. Leung, "A survey on security threats and defensive techniques of machine learning: A data driven view," IEEE access, vol. 6, pp. 12103-12117, 2018.

[59] Manwani, N. and P. S. Sastry (2013) "Noise tolerance under risk minimization," *IEEE Transactions on Cybernetics*, 43(3), pp. 1146–1151.

[60] Behzadan, V. and A. Munir (2017) “Vulnerability of Deep Reinforcement Learning to Policy Induction Attacks,” arXiv preprint arXiv:1701.04143.

[61] Szegedy, C., W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus (2014) “Intriguing properties of neural networks,” in *International Conference on Learning Representations.*

[62] N. Akhtar and A. Mian, "Threat of adversarial attacks on deep learning in computer vision: A survey," IEEE Access, vol. 6, pp. 14410-14430, 2018.

[63] Ateniese, G., G. Felici, L.V. Mancini, A. Spognardi, A. Villani, and D. Vitali. “Hacking smart machines with smarter ones: How to extract meaningful data from machine learning classifiers.” arXiv preprint arXiv:1306.4447 (2013).

[64] Fredrikson, Matthew, Eric Lantz, Somesh Jha, Simon Lin, David Page, and Thomas Ristenpart. “Privacy in pharmacogenetics: An end-to-end case study of personalized warfarin dosing.” In 23rd USENIX Security Symposium (USENIX Security 14), pp. 17-32. 2014.

[65] Wang, B., Y. Yao, B. Viswanath, H. Zheng, and B. Y. Zhao, “With Great Training Comes Great Vulnerability: Practical Attacks against Transfer Learning,” 27th USENIX Security Symposium, 2018, pp. 1281–1297.

[66] Papernot, Nicolas, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. “Practical black-box attacks against machine

learning.” In Proceedings of the 2017 ACM on Asia conference on computer and communications security, pp. 506-519. ACM, 2017.

[67] Gilmer, Justin, Ryan P. Adams, Ian Goodfellow, David Andersen, and George E. Dahl. “Motivating the Rules of the Game for Adversarial Example Research.” arXiv preprint 1807.06732 (2018)

[68] Papernot, N., P. McDaniel, I. Goodfellow, S. Jha, Z. B. Celik, and A. Swami (2016) “Practical Black-Box Attacks against Deep Learning Systems using Adversarial Examples,” arXiv preprint arXiv:1602.02697.

[69] B. Biggio and F. Roli, "Wild patterns: Ten years after the rise of adversarial machine learning," Pattern Recognition, vol. 84, pp. 317-331, 2018.

[70] Hinton, G., O. Vinyals, and J. Dean (2014) “Distilling the knowledge in a neural network,” in *NIPS-14 Workshop on Deep Learning and Representation Learning*, arXiv:1503.02531.

[71] Yuan, Xiaoyong, Pan He, Qile Zhu, and Xiaolin Li, “Adversarial Examples: Attacks and Defenses for Deep Learning.” IEEE Transactions on Neural Network Learning Systems, 2019, pp. 1–20

[72] Huang, S., N. Papernot, I. Goodfellow, Y. Duan, and P. Abbeel (2017) “Adversarial attacks on neural network policies,” arXiv preprint arXiv:1702.02284.

[73] Biggio, B., I. Corona, D. Maiorca, B. Nelson, N. Srndi ˇ c´, P. Laskov, G. Giacinto, and F. Roli (2013) “Evasion attacks against machine learning at test time,” in *Machine Learning and Knowledge Discovery in Databases*, Springer, pp. 387–402.

[74] Papernot, N., P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami (2016) “The Limitations of Deep Learning in Adversarial Settings,” in *1st IEEE European Symposium on Security and Privacy.*

[75] A. Mądry and L. Schmidt, "A Brief Introduction to Adversarial Examples," Gradient Science, [Online]. Available: http://gradientscience.org/intro_adversarial/. [Accessed 19 July 2019]

[76] Biggio, Battista, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. “Evasion attacks against machine learning at test time.” In Joint European conference on machine learning and knowledge discovery in databases, pp. 387-402. Springer, Berlin, Heidelberg, 2013.

[77] Liu, D. C. and J. Nocedal (1989) “On the limited memory BFGS method for large scale optimization,” *Mathematical programming*, 45(1-3), pp. 503–528.

[78] Goodfellow, I. J., J. Shlens, and C. Szegedy (2015) “Explaining and Harnessing Adversarial Examples,” in *3rd International Conference on Learning Representations.*

[79] Moosavi-Dezfooli, S.-M., A. Fawzi, and P. Frossard (2016) "DeepFool: A simple and accurate method to fool deep neural networks," in *IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2574–2582.

[80] Huang, R., B. Xu, D. Schuurmans, and C. Szepesvari (2015) "Learning with a strong adversary," arXiv preprint arXiv:1511.03034

[81] Carlini, N. and D. Wagner (2017) "Towards evaluating the robustness of neural networks," in *IEEE Symposium on Security and Privacy*, pp. 39–57.

[82] Lowd, D. and C. Meek (2005) "Adversarial learning," in *Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining,* ACM, pp. 641–647.

[83] Nelson, B., B. I. Rubinstein, L. Huang, A. D. Joseph, S. J. Lee, S. Rao, and J. Tygar (2012) "Query strategies for evading convex-inducing classifiers," *Journal of Machine Learning Research*, 13, pp. 1293–1332.

[84] Xu, W., Y. Qi, and D. Evans (2016) "Automatically evading classifiers: A Case Study on PDF Malware Classifiers," in *Network and Distributed Systems Symposium.*

[85] Srndi ˇ c, N. ´ and P. Laskov (2014) "Practical evasion of a learning-based classifier: A case study," in IEEE Symposium on Security and Privacy, pp. 197–211.

[86] Vorobeychik, Y. and B. Li (2014) “Optimal randomized classification in adversarial settings,” in *13th International Conference on Autonomous Agents and Multi-Agent Systems*, pp. 485–492.

[87] Papernot, N., P. McDaniel, X. Wu, S. Jha, and A. Swami (2016) “Distillation as a Defense to Adversarial Perturbations against Deep Neural Networks,” in Proceedings of the 37th IEEE Symposium on Security and Privacy, IEEE.

[88] Cquestions.com. C programming interview questions and answers, 2015.

[89] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing Functions in Binaries with Neural Networks. In Proceedings of the 24th USENIX Conference on Security Symposium (SEC’15). USENIX Association, Berkeley, CA, USA, 611-626. http://dl.acm.org/citation.cfm?id=2831143.2831182

[90] A. V. Vishnyakov, “Classification of ROP gadgets”, Proceedings of ISP RAS, 28:6 (2016), 27–36

[91] Grosse, K., N. Papernot, P. Manoharan, M. Backes, and P. McDaniel (2017) “Adversarial Perturbations Against Deep Neural Networks for Malware Classification,” in 22nd European Symposium on Research in Computer Security.

[92] Anderson, H. S., Kharkar, A., Filar, B., & Roth, P. (2017). Evading machine learning malware detection. *Black Hat*.

[93] Chen, L., Ye, Y., & Bourlai, T. (2017, September). Adversarial machine learning in malware detection: Arms race between evasion attack and defense. In *2017*

European Intelligence and Security Informatics Conference (EISIC) (pp. 99-106). IEEE.

[94] Canavese, Daniele & Regano, Leonardo & Basile, Cataldo & Viticchié, Alessio. (2017). Estimating Software Obfuscation Potency with Artificial Neural Networks. 193-202. 10.1007/978-3-319-68063-7_13.

[95] Sebastian Banescu, Christian Collberg, and Alexander Pretschner. 2017. Predicting the resilience of obfuscated code against symbolic execution attacks via machine learning. In *Proceedings of the 26th USENIX Conference on Security Symposium (SEC'17).* USENIX Association, USA, 661–678.

[96] C. Parker, J. McDonald, and D. Damopoulos, "Machine learning classification of obfuscation using Image Visualization," *Proceedings of the 18th International Conference on Security and Cryptography*, 2021.

[97] Albawi, S., Mohammed, T. A., and Al-Zawi, S. (2017). Understanding of a convolutional neural network. In 2017 International Conference on Engineering and Technology (ICET), pages 1–6. Ieee.

[98] Sainath, T. N., Mohamed, A.-r., Kingsbury, B., and Ramabhadran, B. (2013). Deep convolutional neural networks for lvcsr. In 2013 IEEE international conference on acoustics, speech and signal processing, pages 8614–8618. IEEE.

[99] Kabanga, E. K. and Kim, C. H. (2017). Malware images classification using convolutional neural network. Journal of Computer and Communications, 6(1):153–158.

[100] Kalash, M., Rochan, M., Mohammed, N., Bruce, N. D.,Wang, Y., and Iqbal, F. (2018). Malware classification with deep convolutional neural networks. In 2018 9th IFIP international conference on new technologies, mobility and security (NTMS), pages 1–5. IEEE.

[101] J. Salwan, "Storm: Ropgadget - gadgets finder and auto-roper," shell. [Online]. Available: http://shell-storm.org/project/ROPgadget/. [Accessed: 10-Mar-2022].

[102] A. Wibowo Haryanto, E. Kholid Mawardi and Muljono, "Influence of Word Normalization and Chi-Squared Feature Selection on Support Vector Machine (SVM) Text Classification," 2018 International Seminar on Application for Technology of Information and Communication, 2018, pp. 229-233, doi: 10.1109/ISEMANTIC.2018.8549748.

[103] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. D. McDaniel, "Adversarial perturbations against deep neural networks for malware classification," CoRR, vol. abs/1606.04435, 2016. [Online]. Available: http://arxiv.org/abs/1606.04435

[104] W. Hu and Y. Tan, "Black-box attacks against RNN based malware detection algorithms," CoRR, vol. abs/1705.08131, 2017. [Online]. Available: http://arxiv.org/abs/1705.08131

[105] ——, “Generating adversarial malware examples for black-box attacks based on GAN,” CoRR, vol. abs/1702.05983, 2017. [Online]. Available: http://arxiv.org/abs/1702.05983

[106] D. Park, H. Khan and B. Yener, "Generation & Evaluation of Adversarial Examples for Malware Obfuscation," 2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA), 2019, pp. 1283-1290, doi: 10.1109/ICMLA.2019.00210.

[107] D. Park, H. Powers, B. Prashker, L. Liu and B. Yener, "Towards Obfuscated Malware Detection for Low Powered IoT Devices," 2020 19th IEEE International Conference on Machine Learning and Applications (ICMLA), 2020, pp. 1073-1080, doi: 10.1109/ICMLA51294.2020.00173.

[108] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, “Rethinking the inception architecture for computer vision,” in The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), June 2016.

[109] Yang, C., Xu, J., Liang, S. et al. DeepMal: maliciousness-Preserving adversarial instruction learning against static malware detection. Cybersecur 4, 16 (2021). https://doi.org/10.1186/s42400-021-00079-5

[110] C. Ebert, J. Cain, G. Antoniol, S. Counsell and P. Laplante, "Cyclomatic Complexity," in IEEE Software, vol. 33, no. 6, pp. 27-29, Nov.-Dec. 2016, doi: 10.1109/MS.2016.147.

BIOGRAPHICAL SKETCH

Name of Author: Colby B. Parker

Graduate and Undergraduate Schools Attended:

University of South Alabama, Mobile, Alabama

Degrees Awarded:

Doctor of Philosophy in Computing, 2022, Mobile, Alabama

Master of Science in Computer and Information Sciences, 2018, Mobile, Alabama

Bachelor of Science in Computer Science, 2017, Mobile, Alabama

Awards and Honors:

Upsilon Pi Epsilon Honor Society

Scholarship for Service

Publications:

C. Parker, J. T. McDonald, T. Johnsten and R. G. Benton, "Android Malware Detection Using Step-Size Based Multi-layered Vector Space Models," 2018 13th International Conference on Malicious and Unwanted Software (MALWARE), Nantucket, MA, USA, 2018, pp. 1-10, doi: 10.1109/MALWARE.2018.8659372.

C. Parker, J. McDonald, and D. Damopoulos, “Machine learning classification of obfuscation using Image Visualization,” *Proceedings of the 18th International Conference on Security and Cryptography*, 2021.