Improving Adversarial Attacks Against MalConv

Justin Burr

Follow this and additional works at: https://scholar.dsu.edu/theses

Part of the Information Security Commons, OS and Networks Commons, Other Computer Sciences Commons, Software Engineering Commons, and the Systems Architecture Commons
IMPROVING ADVERSARIAL ATTACKS AGAINST MALCONV

A dissertation proposal submitted to Dakota State University in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in

Cyber Operations

March 2022

By

Justin Burr

Dissertation Committee:

Dr. Shengjie Xu
Dr. Austin O’Brien
Dr. Josh Stroschein
DISsertation Approval Form

This dissertation is approved as a credible and independent investigation by a candidate for the Doctor of Philosophy degree and is acceptable for meeting the dissertation requirements for this degree. Acceptance of this dissertation does not imply that the conclusions reached by the candidate are necessarily the conclusions of the major department or university.

Student Name: Justin Burr

Dissertation Title: Improving Adversarial Attacks Against Malconv

Dissertation Chair/Co-Chair: Shengjie Xu
Name: Shengjie Xu
Date: April 25, 2022

Dissertation Chair/Co-Chair: 
Name:

Committee member: Josh Stroschein
Name: Josh Stroschein
Date: April 25, 2022

Committee member: Austin O'Brien
Name: Austin O'Brien
Date: April 25, 2022

Committee member: 
Name:

Committee member: 
Name:

Original to Office of Graduate Studies and Research
Acid-free copies with written reports to library
ACKNOWLEDGMENT

First and foremost, I’d like to thank my family for their love and support throughout the many years I spent completing graduate-level coursework. Pursuing a doctoral degree has been extremely rewarding, but it came at the cost of a lot less quality family time. My beautiful wife Courtney has been incredibly supportive throughout the entire process, and I will be forever grateful. I’d also like to thank my one-year-old baby girl Lorelai for handing out smiles whenever I was feeling overwhelmed. I’m looking forward to spending a lot more time with them now.

I’d also like to thank my dissertation chair, Dr. Shengjie Xu, for all the time he spent teaching and mentoring me throughout the dissertation process. He was an amazing resource for me while conducting this research. I’m thankful for all the time he invested in me and for helping me to complete this program.

The rest of my committee was terrific as well. I’d like to thank Dr. Austin O’Brien and Dr. Josh Stroschein for their valuable insights and feedback throughout the entire process. Josh was also the instructor for several of my core classes in software exploitation, reverse engineering, and malware analysis. I learned so much from him and always found his teaching style to be incredibly effective.
This dissertation proposes several improvements to existing adversarial attacks against MalConv, a raw-byte malware classifier for Windows PE files. The included contributions greatly improve the success rates and performance of gradient-based file overlay attacks. All improvements are included in a new open-source attack utility called BitCamo.

Several new payload initialization strategies for use with gradient-based attacks are proposed and evaluated as potential replacements for the randomized initialization method used by current attacks. An algorithm for determining the optimal payload size is also proposed. The resulting improvements achieve a 100% evasion rate against eligible target executables using an average payload size of only 300 bytes. The results are substantially better than those reported by other open-source tools or attacks proposed within the research literature.

Existing gradient attacks against MalConv contain a long-running byte reconstruction phase necessary to map backwards across a non-differentiable embedding layer used by the model. Three proposals are presented to significantly improve the runtime of this phase, including the addition of parallelism, limiting the scope of reconstruction to the payload only, and introducing a K-D tree data structure to allow for blazing fast spatial searches in comparison to the L_2 distance metric used by current attacks.

A pre-detection mechanism proposed in previous research checks if executables have the same code section hash but a different overall hash with respect to known malicious files, allowing adversarial examples to be immediately rejected by a detection pipeline before MalConv evaluates the sample. This dissertation proposes a single-byte code section attack that can completely bypass this defense mechanism in over 63% of samples. The pre-detection attack can be used in conjunction with the other new improvements to offer a formidable attack capability against MalConv and other detection models sharing a similar architecture.
DECLARATION

I hereby certify that this dissertation constitutes my own product, that where the language of others is set forth, quotation marks so indicate, and that appropriate credit is given where I have used the language, ideas, expressions, or writings of another.

I declare that the dissertation describes original work that has not previously been presented for the award of any other degree of any institution.

Signed,

[Signature: Justin Burr]

Justin Burr
# TABLE OF CONTENTS

ACKNOWLEDGMENT .......................................................................................................................... III

ABSTRACT ........................................................................................................................................ IV

DECLARATION ....................................................................................................................................... V

INTRODUCTION ...................................................................................................................................... 1

- INTRODUCTION TO THE PROBLEM .............................................................................................. 1
- AN INTRODUCTION TO ADVERSARIAL EXAMPLES USING IMAGES ............................................. 2
- ADVERSARIAL EXAMPLES FOR MALWARE CLASSIFIERS ............................................................... 4
- INTRODUCTION TO MalConv ........................................................................................................... 5
- EFFECT OF PACKING ON MalConv .................................................................................................. 6
- ADVERSARIAL ATTACKS AGAINST MalConv .................................................................................. 7
- Predetection Pipelines ......................................................................................................................... 8

STATEMENT OF THE PROBLEM ......................................................................................................... 8

RESEARCH QUESTIONS ....................................................................................................................... 9

SIGNIFICANCE OF THE RESEARCH ................................................................................................... 9

ORGANIZATION OF THIS PAPER ....................................................................................................... 11

LITERATURE REVIEW ......................................................................................................................... 12

- INTRODUCTION ................................................................................................................................. 12
- WINDOWS PORTABLE EXECUTABLE (PE) FORMAT ............................................................................. 12
- MACHINE LEARNING AND DEEP LEARNING .................................................................................... 20
- ADVERSARIAL EXAMPLES .................................................................................................................. 24
- MALWARE DETECTION ....................................................................................................................... 27
- DEEP LEARNING FOR MALWARE DETECTION .................................................................................. 29
- PRACTICAL ADVERSARIAL ATTACKS AGAINST MalConv ................................................................. 34
- ADVERSARIAL DEFENSES FOR PE FILES .......................................................................................... 38

RESEARCH METHODOLOGY ................................................................................................................ 41

- STATISTICAL DIFFERENCE-MAKING EXPERIMENTS ....................................................................... 42
- STAKEHOLDERS ................................................................................................................................. 43
- RESEARCH QUESTIONS AND HYPOTHESES ................................................................................... 43
- NOVEL CONTRIBUTIONS AND PROPOSED ATTACK IMPROVEMENTS ............................................... 44
- TOOLS AND ENVIRONMENT .............................................................................................................. 48
- ARTIFACT DESIGN: IMPLEMENTATION OF APPEND-BASED FGSM ATTACK ................................. 53
- ARTIFACT DESIGN: PAYLOAD INITIALIZATION TECHNIQUES ......................................................... 56
- ARTIFACT DESIGN: DEFEATING THE PRE-DETECTION MECHANISM ................................................ 58
ARTIFACT DESIGN: CLI OUTPUT .................................................................62
DATASET SELECTION .................................................................................64
LIMITATIONS AND ASSUMPTIONS .........................................................66
EXPECTED CONTRIBUTIONS ....................................................................67
SUMMARY .................................................................................................68

RESULTS ......................................................................................................69

PAYLOAD INITIALIZATION STRATEGIES ..................................................69
EVADING THE PRE-DETECTION MECHANISM .........................................80
PERFORMANCE IMPROVEMENTS TO THE BYTE RECONSTRUCTION PHASE ....................................................81
VALIDATION OF K-D TREE QUERY ACCURACY .......................................83
ACHIEVING HIGH EVASION RATES AND MINIMIZING PAYLOAD SIZES ..........................................................84
COMPARISON TO EXISTING ATTACKS ......................................................86

CONCLUSION ............................................................................................91

CONTRIBUTIONS .....................................................................................91
RESEARCH QUESTIONS ...........................................................................92
APPLICABILITY OF CONTRIBUTIONS TO OTHER MODELS ......................94
APPLICABILITY OF MODERN ADVERSARIAL ATTACKS TO MALCONV .................................95
LIMITATIONS AND FUTURE RESEARCH ..................................................96

REFERENCES ............................................................................................99
LIST OF EQUATIONS

Equation 1. Fast Gradient Sign Method (FGSM). ...................................................... 24
Equation 2. One-Step Target Class Method................................................................. 26
LIST OF TABLES

Table 1. A comparison of payload initialization strategies................................. 70
Table 2. A comparison of the top eight payload initialization strategies............... 79
Table 3. Evasion rates of the pre-detection mechanism..................................... 80
Table 4. Performance comparison of reconstruction methods............................ 82
Table 5. Evaluation of attack success using 0xBF (191) byte initialization............ 84
Table 6. Comparison of BitCamo v1.1.0 to SecML Malware v0.2.4. ...................... 87
LIST OF FIGURES

Figure 1. Example of an adversarial example, inspired by Szegedy et al. (2013) ........ 2
Figure 2. Modifying a single pixel within in a picture of a beach. ......................... 3
Figure 3. Visualization of the adversarial attack proposed by Kreuk et al. (2018) ....... 7
Figure 4. Structure of a Windows PE file, adapted from Andriesse (2018) ............. 14
Figure 5. Slack space appearing after the code section in notepad.exe .................... 16
Figure 6. Payload locations in PE files, inspired by Demetrio et al. (2021b) ............. 18
Figure 7. A modern perceptron, as originally conceived by Rosenblatt (1958) ......... 21
Figure 8. A simple MLP neural network used for classification. ......................... 22
Figure 9. Demonstration of a max pooling layer. ............................................ 23
Figure 10. Difficulty of adversarial attacks, adapted from Chakraborty et al. (2018). 26
Figure 11. ML malware detection workflow, adapted from Ling et al. (2021) .......... 29
Figure 12. MalConv architecture, adapted from Raff et al. (2018a) ....................... 31
Figure 13. Comparison of CNN architectures, adapted from Krcál et al. (2018) ...... 33
Figure 14. Relationship between preservation goals, inspired by Ling et al. (2021) .. 34
Figure 15. MalConv attack algorithm proposed by Kolosnjaji et al. (2018) .......... 35
Figure 16. MalConv attack algorithm proposed by Kreuk et al. (2018) ................. 36
Figure 17. The pre-detection mechanism, adapted from Chen et al. (2019) ........... 40
Figure 18. Visualization of a K-D tree in two dimensions. ................................. 47
Figure 19. Keras code defining MalConv, adapted from Anderson & Roth (2018) ... 52
Figure 20. A Python implementation of the append-based FGSM attack. ............. 54
Figure 21. Python function used to find the closest byte to an 8D-embedding. ....... 55
Figure 22. Implementation of simple payload initialization strategies ................. 57
Figure 23. Implementation of the weighted payload initialization strategy .......... 58
Figure 24. Binary patching code used to defeat the pre-detection mechanism ....... 60
Figure 25. Using LIEF to dynamically determine the code section .................... 61
Figure 26. IDA Pro view of the valid instructions captured by Capstone ............... 62
Figure 27. Output of the attack tool during the reconstruction phase ................. 63
Figure 28. Output of the attack tool, post-reconstruction. ................................. 63
CHAPTER 1

INTRODUCTION

Introduction to the Problem

Artificial intelligence (AI) has become a topic of immense popularity and importance over the past decade, producing intense media hype and capturing the attention and imagination of the public. A branch of AI known as machine learning (ML) excels in its ability to recognize patterns, make predictions, and perform tasks without being explicitly programmed. Machine learning is frequently applied to problems in the cybersecurity domain, often to detect anomalous activities (Xu, Qian, & Hu, 2017, 2019a, 2019b, 2019c, 2020). Over the course of the past decade, the explosion of big data and advances in graphical processing units (GPUs) have allowed an ML technique known as deep learning (DL) to thrive (Chollet, 2018). Deep learning was originally inspired by the human brain and uses artificial neural networks (ANN) to learn higher-level features from large quantities of raw input.

The use of deep neural networks (DNNs) has made it possible to perform tasks such as image classification, object detection, speech recognition, natural language processing, and even malware detection, the latter of which will become the central theme in this dissertation. Neural networks are not without their share of security concerns though. Szegedy et al. (2013) first demonstrated that neural networks are vulnerable to an attack known as adversarial examples, where attackers can make small modifications to the input of a neural network with the goal of causing the network to make an incorrect prediction or classification.

To best explain adversarial examples, the next section will begin by introducing the concept in the context of image classification, both to provide historical context and allow for ease of explanation. The chapter will then transition to discuss the application of adversarial examples within the malware domain. This sets the stage for the goal of this dissertation, where attacks will be developed against a popular neural network called MalConv, which uses the raw bytes of a Windows Portable Executable (PE) file to determine maliciousness (Raff et al., 2018a). Success rates of existing attacks against MalConv appear to be wildly inconsistent.
throughout the research literature, with many attacks entailing a high computational cost. This design science dissertation seeks to determine if improvements to accuracy or performance are possible, and to include them in an open-source attack tool. Specific proposals for improvements are presented at the end of this chapter after a more thorough introduction to the topic and explanation of current issues.

**An Introduction to Adversarial Examples Using Images**

First observed by Szegedy et al. (2013), an adversarial example is an input to a machine learning model in which an attacker has made subtle modifications with the goal of causing a misclassification to occur. Figure 1 shows an example in which an image of a beach can be selectively modified so that the target machine learning model now classifies it as a mountain range with high confidence, while still appearing as a beach to human observers.

![Figure 1. Example of an adversarial example, inspired by Szegedy et al. (2013).](image)

When creating adversarial examples for use with image classifiers, any pixel within the input image can be freely modified by the attacker. The only constraint is that their final values must remain within the valid range allowed by the associated image format, for example, 0-255 for RGB images. No restrictions exist on which pixels are allowed to be modified, or how many are allowed to be modified, but it's desirable for the change to go unnoticed by human observers. Figure 2 illustrates that an image is merely a collection of pixels in which any pixel can be freely modified to produce another valid image.
For an adversarial attack to succeed, the attacker must also figure out which pixels should be changed and what values those pixels should contain to fool the machine learning model into producing a misclassification. In a white-box attack scenario, attackers have full access to all information regarding a trained classification model, to include the model’s parameters and weights (Yuan, He, Zhu, & Li, 2019). In these scenarios, adversarial examples can be created by first computing a gradient (Yuan, et al., 2019). A gradient is a vector of partial derivatives that point in the direction of steepest ascent with respect to an input function. Adversarial examples use gradients to either maximize a loss function with respect to the actual classification or minimize a loss function with respect to a target classification. Loss is a measurement for how well a model is performing in terms of how far off the model’s predictions are from the true labels of the input data. A more thorough explanation of loss, gradients, and the mathematical underpinnings of machine learning will be provided in chapter 2.

The earliest formalization of gradient-based attacks is the Fast Gradient Sign Method (FGSM), originally proposed by Goodfellow, Shlens, & Szegedy (2014). The attack works by calculating the gradient with respect to the input, and then slightly nudging the input in the direction of the gradients as to maximize the overall loss (Goodfellow et al., 2014). When used to train a model, gradients are instead used to update the weights of a network instead of the inputs. When FGSM is used, however, the weights remain fixed and are never changed. A more thorough explanation of FGSM is provided in chapter 2.

In black-box scenarios, an attacker has no access to the internals of a target machine learning model but is generally allowed to inspect the final output (Yuan, et al., 2019). Although computation of gradients is not possible in black-box scenarios, a common
technique is to perform a gradient-based attack on a white-box model trained with similar data and then attempt to transfer the resulting adversarial example back to the black-box model (Yuan, et al., 2019). These models are referred to as substitute or surrogate models and can often be trained by repeatedly querying the black-box model to produce a series of input-output pairs (Chakraborty, Alam, Dey, Chattopadhyay, & Mukhopadhyay, 2018). The transferability of adversarial examples makes white-box attacks relevant in scenarios where no information about the target machine learning model is known. The construction of substitute models is a form of extraction attack, but is often computationally expensive to complete.

**Adversarial Examples for Malware Classifiers**

The previous section provided a gentle introduction to adversarial examples by using image classifiers as an example. Adversarial examples are in no way limited to the image domain, however, and can be applied to a wide variety of applications. This dissertation will focus exclusively on adversarial examples for malware classifiers using Windows PE files.

Producing adversarial malware is a significantly more challenging problem than producing adversarial images. Recall that any pixel within an image can be freely modified to take on a wide range of possible values. The same cannot be said about executable file formats, as altering bytes has the potential to change the program’s functionality or its ability to run altogether. This is because the content of a Windows PE file includes important headers, pointers to other parts of the file, and even the compiled code used by the program when it runs. A more thorough review of the Windows PE file format will be discussed in chapter 2.

Research into adversarial attacks against malware classifiers was historically very theoretical in nature, producing representative feature vectors instead of fully runnable programs (Park & Yener, 2020). This reiterates the tremendous difficulty of producing working attacks within this area of research.

As previously mentioned, not all parts of a Windows PE file can be freely modified without fear of breaking the program’s functionality. Most gradient-based adversarial attacks therefore tend to overcome these challenges by locating positions within the file where a large, contiguous series of bytes can be placed without disrupting functionality. Prime
examples include appending data to the very end of the file, utilizing unused space at the end of a section, or even adding a new section. Other unused space exists but is often more restrictive in terms of the payload size capable of being placed at those locations. Demetrio et al. (2021b) propose several methods oriented towards manipulating Windows PE headers. A thorough review of the usable space within Windows PE files can be found in chapter 2, along with relevant binary patching techniques.

**Introduction to MalConv**

Traditionally, most machine learning models for malware detection were trained using a feature engineering approach, where various properties and characteristics about the malware would be gathered and used as input to the model (Park & Yener, 2020). This includes static features such as op code frequencies or dynamic features such as frequency of system calls (Park & Yener, 2020).

This dissertation will focus exclusively on attacking a model that uses a fundamentally different technique. MalConv is a neural network using a file’s raw bytes as input, effectively eliminating the need for a feature selection process and the domain expertise it tends to require (Raff et al., 2018a). Other key advantages to this model include its strong ability to generalize and to detect malicious features regardless of their specific location within the file (Raff et al., 2018a). A thorough analysis of the MalConv architecture is provided in chapter 2.

This dissertation opts to use a pre-trained version of MalConv, trained on the EMBER dataset containing 1.1 million binary files (Anderson & Roth, 2018). This model differs slightly from the one proposed on the original MalConv specifications, with the most important change being the use of 1 MB inputs instead of 2 MB to overcome memory limitations on GPUs (Anderson & Roth, 2018). The decision to use the EMBER-trained MalConv model was made for the following reasons:

- The effort required to collect and maintain a representative dataset this large is a large project in and of itself.
- It was one of the only pre-trained, open-source MalConv models publicly available on the internet.
• The authors had significantly more compute power available to them. Anderson & Roth (2018) note that the model took 10 days to train using dual Titan X (Pascal) GPUs.

• Many other researchers opt to use this pre-trained model as well (Demetrio, Biggio, Lagorio, Roli, & Armando, 2019).

Effect of Packing on MalConv

Due to this dissertation’s focus on offensive-oriented techniques, no attention will be given to defensive improvements to the model itself at this time. Packing and obfuscation are other well-known techniques that can be used to evade malware classifiers, to include MalConv. Packers are a type of software that can be used to re-write a target program by compressing it within another program. Packers are sometimes used by benign software to protect intellectual property or reduce file size (Demetrio et al., 2019). As it pertains to machine learning-based malware classifiers, packers have the effect of hiding malicious features.

Demetrio et al. (2019) explain that packing is a much more invasive procedure than adversarial attacks, and do not clearly demonstrate the vulnerability of machine learning models to extremely small changes. Many security tools also employ automated unpacking routines, which has been an extremely active area of research for well over a decade (Coogan, Debray, Kaochar, & Townsend, 2009). Adversarial attacks are therefore being researched as an alternative approach to packing. In some cases, the two techniques may even complement each other. For example, a detection pipeline may opt to unpack a sample and send the resulting unpacked executable into a machine learning model. If the attacker applies adversarial perturbations before packing their program, it may be possible to evade detection.

From an offensive perspective, the adversarial attacks used in this dissertation can be applied to packed samples in much the same way as non-packed samples. As will later be demonstrated, a packed malware sample receiving a positive result from MalConv can still be perturbed until it achieves evasion, all without disrupting functionality. The attack tool developed in this dissertation will be tested using samples drawn uniformly at random from a large dataset containing over 10 million malicious binaries, including both packed and unpacked binaries (Harang & Rudd, 2020).
**Adversarial Attacks Against MalConv**

Kolosnjaji et al. (2018) and Kreuk et al. (2018) produced amongst the earliest attacks against MalConv. Each uses a gradient-based attack in which the payload is appended to the very end of the file, and each point to MalConv’s non-differentiable embedding layer as being a central issue they must overcome. The embedding layer acts as a lookup table, where each byte is mapped to an 8-dimensional array (Raff et al., 2018a). This presents problems for gradient-based attacks, being that the embedding layer is non-differentiable.

Kreuk et al. (2018) solves this problem by running the input bytes through the embedding layer only, producing an embedded representation of the input bytes. The authors then run the previously described FGSM attack on the remainder of the model with respect to the embedded bytes only. After the embedded bytes have been modified, the authors then attempt to map backwards through the embedding layer, locating the closest possible byte for each of the corresponding embedded representations. This process is also referred to as the reconstruction phase and is somewhat computationally expensive in comparison to the rest of the attack. A visualization of this attack can be seen in Figure 3.

![Figure 3. Visualization of the adversarial attack proposed by Kreuk et al. (2018).](image)

Due to the inability to modify many parts of a PE file, Kreuk et al. (2018) initialized a fixed-length payload at the very end of the original file. The authors opted to initialize this payload using bytes selected uniformly at random. Research to determine if this phase of the attack can be improved will become a central focus for this dissertation. A more in-depth
examination of existing attacks, to include the gradient-based attacks Kreuk et al. (2018) and Kolosnjaji et al. (2018), can be found in chapter 2.

**Predetection Pipelines**

Although the raw byte inputs used by MalConv eliminate the need for domain knowledge, it appears likely that future research will attempt to reincorporate it back into the process to some degree. Demetrio et al. (2021b) suggest that future research directions should explore the addition of domain knowledge into the learning process to bridge the gap between classifiers that are completely feature-driven and classifiers that use a raw-byte approach. Park & Yener (2020) suggest that future research directions should include detection pipelines instead of just using a single machine learning model.

Chen et al. (2019) proposed a pre-detection mechanism capable of rejecting 100% of adversarial examples in their initial experiment. The pre-detection mechanism works by computing a hash of both the entire PE file and of the code section specifically, comparing the hashes to those contained within a malware database. If the code section hash of a new sample matches the code section hash of a known malicious file but their overall file hashes differ, the authors conclude some other part of the file has been tampered with. The sample is then immediately rejected as being an adversarial example. A detailed description of their proposal can be found in chapter 2.

**Statement of the problem**

It remains unclear if the success rates from the gradient-based append attacks proposed by Kolosnjaji et al. (2018) and Kreuk et al. (2018) can be reliably reproduced, as other researchers appear to report lower success rates when using these techniques on models trained with other datasets (Suciu, Coull, & Johns, 2019; Chen et al., 2019). Further examination is needed to determine if the inability to reproduce the initial results is entirely due to dataset discrepancies, or if implementation issues existed as well.

Chen et al. (2019) believes the initialization of gradient-based payloads using random bytes is likely a very poor choice. However, few payload initialization strategies are proposed
within the research literature. Initializing the payload with a new and improved strategy could have a noticeable impact on overall success rates of adversarial attacks.

The byte reconstruction phase of the attack proposed by Kreuk et al. (2018) appears to be computationally expensive. An improvement to this phase of the attack would allow for significantly more samples to be tested during experiments, thereby improving the quality of future research.

Chen et al. (2019) proposed a pre-detection mechanism that has an obvious avenue for attackers to target. Any change to the code section would theoretically allow an attacker to completely bypass the pre-detection mechanism. An attacker merely needs to change a single byte to produce different code section hash. Confirmation that this defensive resource can be easily defeated is the final problem this dissertation will attempt to answer.

Research Questions

This following research questions will be answered:

1. Can the append-based FGSM attack be re-implemented and improved to achieve consistently high evasion rates?
2. Can the append-based FGSM attack benefit from a new byte initialization strategy?
3. Can performance of the long-running reconstruction phase of the append-based FGSM attack be improved?
4. Can the recently proposed pre-detection mechanism be completely bypassed?

Significance of the research

Adversarial attacks may give the impression of being theoretical in nature, with many successful attacks occurring within controlled test environments. Real-world occurrences have already been observed though. A research team from Skylight Cyber reverse engineered Cylance’s endpoint protection product and discovered a vulnerability allowing 90% of malware to evade detection (Ashkenazy & Zini, 2019). Cylance is a company well known for their heavy use of AI and ML to detect malware. The researchers tricked the Cylance detection model by appending benign strings from the video game Rocket League, effectively creating a universal bypass (Ashkenazy & Zini, 2019).
Large software companies are also beginning to recognize adversarial machine learning attacks as being a serious problem. A search of recent job postings reveals several large software companies such as Microsoft and NVIDIA are hiring researchers and AI red team operators to begin evaluating their products for susceptibility to these attacks.

This dissertation is an offensive-oriented project with the goal of improving attacks against malware classifiers, and to completely bypass a recently proposed pre-detection mechanism. Due to the limited number of tools capable of producing adversarial attacks using Windows PE files, the improvements offered within this dissertation will be made public in the form of a command-line program called BitCamo. To further contribute towards efforts of the security research community, the tool will also feature a bulk-processing feature to allow for experiments to be conducted using large datasets.

Research into adversarial malware is important because it provides a fundamentally different way for malware to hide in plain sight, whereas other techniques such as packing are much more invasive, with many security products already attempting to account for them. As demonstrated in the attack against Cylance, adversarial attacks have the potential to be a much more powerful capability as compared to packing, sometimes allowing a universal bypass to occur.

While the development of offensive techniques may seem counter intuitive as an end goal, security is a very much a cat-and-mouse game. A new offensive capability will always be met with a new defensive capability. As such, both offensive- and defensive-oriented research will further the field as whole. By demonstrating the ease at which adversarial malware can be created, it raises awareness to the problem in hopes that security vendors will begin to incorporate additional safeguards in their machine learning-based applications and take adversarial machine learning attacks more seriously.

The release of this dissertation and its associated source code is not expected to cause undue harm to security vendors using machine learning technology within their products. The release of open-source attack software is generally viewed in a positive light by the security industry, as responsible use of these products helps to harden existing defenses and allow companies to simulate real-world attacks (Harang, 2020). For example, the use of tools such as Metasploit and Cobalt Strike has proved to invaluable for use in penetration testing and
offensive security engagements, despite the potential for attackers to use these products as well.

Organization of this paper

In the next chapter, a full literature review is provided, greatly expanding upon each of topics introduced in this chapter. Chapter 3 outlines a plan for how the research will be conducted, to include a description of the research methodology and details for how the proposed attack will be implemented and later validated. Chapter 4 then presents the results of the research and demonstrates the success of the new attack methods. Finally, Chapter 5 provides a summary of the paper, lessons learned, limitations, and potential research directions for future work.
CHAPTER 2

LITERATURE REVIEW

Introduction

This chapter reviews existing literature from a wide variety of topics pertaining to this dissertation. Topics include the Windows PE file format, machine learning, deep learning, adversarial examples, malware detection, MalConv, existing adversarial attacks against MalConv, and defenses against adversarial attacks. Given the tremendous amount of background knowledge required to embark upon this research project, the goal of this chapter is not to provide a comprehensive review of every single one of these subjects, but rather to provide the minimal amount of prerequisite knowledge necessary to understand the remaining chapters of the dissertation.

Windows Portable Executable (PE) Format

This section reviews prior research pertaining to the Windows PE file format. The goal is not to provide a comprehensive review of the format, but rather to provide a brief overview of its structure, convey the difficulty of modifying bytes without breaking functionality, and to identify methods for embedding adversarial bytes within the file. Positions suitable for placement of the attack generally coincide with decades-old research involving data hiding techniques within Windows PE files. Early research in data hiding techniques predates adversarial examples and therefore tends to focus on hiding entire files or additional code within PE files. A later section of this chapter will review literature involving existing adversarial attacks on raw byte malware classifiers, where it will become apparent that data hiding concepts have been extended to work with adversarial examples.

To begin the discussion of Windows PE files, it is necessary to understand what an executable file format is used for. A binary executable is a compiled program containing machine code that a computer can execute (Andriesse, 2018). There are many types of
executable file formats, but Linux systems will generally use the Executable and Linkable Format (ELF), whereas Windows systems use the Portable Executable (PE) format (Andriesse, 2018). This dissertation will focus exclusively on the Windows PE file format, following the same direction as other research projects that attack raw byte malware classifiers. This is a natural progression given that existing defensive-based work has primarily elected to use Windows PE files to train their classification models. This is of no surprise given that most malware is still designed to target users of Windows operating systems. In Q1 2020, an astonishing 83.45% of malware was reported to be targeting Windows devices (AV-TEST, 2020).

Andriesse (2018) explained that the PE format is based off Unix’s Common Object File Format (COFF) and is therefore referred to in some literature as PE/COFF. He continues by describing the historical reasoning for the presence of an MS-DOS header at the beginning of all PE files, which always begin with the ASCII characters “MZ” and are sometimes referred to as “magic bytes”. To assist in the transition from the MS-DOS format to the newer PE format, he notes that all PE files begin with an MS-DOS header and contain a pointer to the beginning of the newer PE header. This allows older loaders to execute PE files, after which they print out “This program cannot be run in DOS mode”, a string contained within the MS-DOS stub. Newer loaders will instead opt to resolve the pointer to the PE header, contained within a field called e_lfanew, allowing the real PE program to begin executing. The MS-DOS header, MS-DOS-stub, and PE header can be seen in Figure 4, along with the rest of the PE file structure. Because the leading “MZ” string and e_lfanew field are the only parts of the MS-DOS header used by newer loaders, a later section of the literature review will demonstrate attacks that attempt to utilize all remaining space within the MS-DOS header to embed adversarial bytes (Demetrio et al., 2021b).
Andriesse (2018) describes three major entities within the PE header: the PE signature, the PE file header, and the PE optional header. The PE signature is merely another series of magic bytes, namely “PE\0\0”. The PE file header contains important fields such as Machine, identifying the target architecture, and others such as NumberOfSections and SizeOfOptionalHeader to assist in parsing the remainder of the file. The last entity Andriesse describes is the PE optional header, which is not actually optional in the case of executables. The PE optional header includes several important pointers and relative virtual addresses (RVAs), such as ImageBase, BaseOfCode, and AddressOfEntryPoint. Finally, Andriesse notes that the export directory and import directory can also be found through RVAs located here.

The last major components to the PE files format are the sections and section headers. Andriesse (2018) states that the headers contain important information such as the
SizeOfRawData, PointerToRawData, VirtualSize, VirtualAddress, and Characteristics. He concludes with a discussion of noteworthy sections and their purpose. The .text section contains assembled code, .rdata contains read-only data, .data contains data that is both readable and writable, .bss contains zero-initialized data, .edata contains exported functions, and .idata contains imported functions.

The literature review of PE files has thus far focused almost exclusively on reviewing well-documented aspects of the file format. Although not mentioned in many texts, PE files can also include an overlay, which is best defined as “any data appended beyond the nominal end of the PE image” (Hnatiw, Robinson, Sheehan, & Suan, 2007). Hnatiw et al. also explain that unlike other sections of the PE file, overlays are not automatically loaded into memory by Windows at execution time, existing only on disk initially. As it pertains to machine learning models whose input is the entire raw bytes of a file, file overlays will still be considered by the model. Kolosnjaji et al. (2018) made one of the first attempts at embedding adversarial bytes within a PE file. Although the authors never use the term “overlay”, their strategy of appending bytes to the end of the file is one of the first known examples of an adversarial attack utilizing file overlays.

The data normally contained within PE overlays need not be homogeneous, and examples of legitimate use cases include certificates, debugging information, and data appended by packing utilities such as UPX (Hnatiw et al., 2007). Hnatiw et al. refer to each of these individual items as overlay regions. Although the mere presence of an overlay is insufficient to determine maliciousness of a file, only 30% of benign PE files contain an overlay, whereas twice as many malicious files contain one (Hnatiw et al., 2007). Finally, as certificates are the most common use case for legitimate PE overlays, binary analysis tools such as PEiD will automatically attempt to extract certificates, making a distinction between certificates and the rest of the overlay.

Another common technique for embedding data within PE files is via slack space. Slack space refers to unused space that may be present at the end of section. This situation arises when the amount of data within a section is less than the section size allocated at compile time (Shin, Kim, Byun & Lee, 2008). Shin et al. used slack space to demonstrate a data hiding technique in which they embedded an encrypted JPEG image within a PE file.
As a more precise definition, slack space is present when a section’s `SizeOfRawData` field is less than the `VirtualSize` field, occurring when compilers round sizes up to align sections on a boundary (Goppit, 2006). As was also the case with overlays, Goppit notes that slack space is unused and is therefore not loaded into memory at execution time. For an attacker to introduce new embedded code that is loaded into memory at runtime, they would also need to increase the `VirtualSize` field accordingly. Finally, Goppit also demonstrates that compilers always fill slack space with null bytes (0x00) by default. This is important to point out for the purpose of introducing adversarial bytes, as the use of any other value within slack space could potentially raise some red flags. It may therefore be advisable to make a corresponding increase to the `VirtualSize` in this situation as well. This effectively reduces the slack space size, making the adversarial bytes appear to be within the actual section data instead. This change is unlikely to break functionality, as no pre-existing code would have a `CALL` or `JMP` instruction destined for this new location. A visualization of slack space can be seen in Figure 5, where the hexdump utility shows a cave of null bytes appearing between the code and data sections of a PE file.

![Figure 5. Slack space appearing after the code section in notepad.exe.](image-url)
If insufficient slack space exists, it is also possible to enlarge an existing section (Goppit, 2006). Goppit suggests that this is only practical to perform with the very last section in the file, however, as the need to adjust pointers makes this task practically impossible to do with other sections without re-compiling. He then provides instructions necessary to extend a section, which include increasing both the \texttt{VirtualSize} and \texttt{SizeOfRawData} fields for the final section, and then increasing the \texttt{SizeOfImage} field within the PE header by the same amount. He mentions that you may want to update the \texttt{SizeOfCode} and \texttt{SizeOfInitializedData} fields within the PE header, depending on which section is being extended. While he suggests that this final step only needs to be performed as a matter of completeness, as the program will still run either way, an attacker may want to ensure that these values are accurate to thwart any defenses that may be performing format validation.

The final way to add a significant amount of space to a PE file is to add additional sections. As updating references throughout the entire file is a nearly impossible task, the new section should be added after the last existing section at the very end of the file. Goppit (2006) documented instructions for this task as well. Within the PE header, he states that the \texttt{SizeOfImage} and \texttt{NumberOfSections} fields must be updated accordingly. He then says that a new section header should also be added after the last existing section header, where there will typically be some space available to add new entries before the actual sections begin. The new section header will need to include the following fields: a section name, \texttt{VirtualSize}, \texttt{VirtualAddress}, \texttt{SizeOfRawData}, \texttt{PointerToRawData}, and \texttt{Characteristics}. Goppit also provides detailed instructions for how to compute each value, with the only unobvious considerations being that the virtual address and raw offsets must be aligned to boundaries.

There are numerous ways to embed data when a much smaller amount of space is required. One such attack involves using the space occupied by unimportant fields within the MS-DOS header, as only the \texttt{e_magic} and \texttt{e_lfanew} fields are needed to load executables on modern systems (Demetrio, Biggio, Lagorio, Roli & Armando, 2019). A research group consisting of many of the same authors later extended this concept to include overwriting bytes within the MS-DOS stub as well (Demetrio et al., 2021b). The authors refer to the original attack as the Partial DOS attack and the latter as a Full DOS attack. The attacks are pictured in Figure 6.
Demetrio et al. (2021b) also propose an attack called Extend, whereby the MS-DOS header is enlarged, creating additional space before the start of the PE header. As was the case with other techniques, it requires section offsets to be adjusted accordingly. The authors also propose an attack called Shift, where new space is created by shifting a section forward. The Shift and Extend attacks are pictured in Figure 6.

![Figure 6](image_url)

Figure 6. Payload locations in PE files, inspired by Demetrio et al. (2021b).

Up to this point, the techniques discussed in this section largely revolve around carving out contiguous chunks of data within the PE file by adding or extending sections, appending a file overlay, or abusing the file format by overwriting data within unnecessary header fields. Another approach is to make a series of functionality preserving operations. Anderson et al. (2018) describes several examples of functionality preserving operations, which comprise the action space of their reinforcement learning-based attack:

- Adding an unused function to the Import Address Table (IAT)
- Adding an unused section, as previously discussed
• Adding space to the end of a section by utilizing slack space, as previously discussed
• Adding space to the end of a file by utilizing overlays, as previously discussed
• Adding a new entry point that jumps to the original entry point
• Modifying section names
• Modifying the header checksum
• Modifying debug info
• Removing signer information
• Packing or unpacking the binary

Finally, a technique that allows for more fine-grained manipulations of a PE file is through code transformations. Song et al. (2020) describes four types of transformations that are used to achieve this goal. First, an instruction can be replaced with a semantically equivalent instruction of the same length. A prime example is adding a negative instead of subtracting a positive. Second, the authors suggest reassigning registers within a function, provided that other code is not dependent on them. For example, instructions using the EBX register could be changed to use the ECX register, and vice-versa. Third, instructions can be re-ordered after careful consideration for dependencies, ideally using a dependency graph. Last, the order that registers are pushed and popped onto the stack across functional calls can be changed. These transformations are an adaptation of a technique known as in-place code randomization, first proposed by Pappas, Polychronakis, & Keromytis (2012) as a defense against return-oriented programming (ROP).

Although seemingly not well documented within published literature on code transformations, a blog post by Woodruff (2020) proposes hiding messages within x86 binaries using a trick known as semantic duals. Woodruff starts by demonstrating the previously discussed concept of semantically equivalent instructions. For example, he shows that the following operation of zeroing out the EAX register can be performed in the following ways:

• xor eax, eax
• and eax, 0
• mov eax, 0
• sub eax, eax
• `lea eax, [0]`

Woodruff (2020) takes this concept one step further by abusing the ModR/M byte, noting that register-to-memory and memory-to-register representations allow for the previous example of `xor eax, eax`, to have two possible encodings, as either `31 c0` or `33 c0`. The author calls this phenomenon a semantic dual, which he defines as different opcodes for the same instruction family that also share the same length, behavior, and performance. He documents a short list of semantic duals that he was able to locate within the X86 instruction set, which include certain variations of the `ADD, ADC, AND, OR, XOR, SUB, SBB, MOV, and CMP` instructions.

**Machine Learning and Deep Learning**

Arthur Samuel first coined the term “machine learning”, where he described a system capable of learning from experience rather than being explicitly programmed (Samuel, 1967). Machine learning systems learn by using data to build decision models. The data used to train the models is referred to as training data, where individual inputs are called training instances or samples. The accuracy of the model is then validated using a testing set. Different datasets are used for training and testing to ensure the model is extracting meaningful patterns as opposed to simply remembering previously observed samples. Machine learning systems can be classified into four categories, largely based on the level of human supervision required to train the model. The categories include:

- Supervised learning
- Unsupervised learning
- Semi-supervised learning
- Reinforcement learning

This dissertation examines a supervised learning technique known as classification, where models are used to predict the class (or label) of new data. In the context of malware classification, the two labels used for the purposes of this research are malicious and benign. When only two labels are present, the problem is also referred to as a binary classification problem. In classic machine learning systems, the model determines the classification using a
set of predictive features. The human-driven process of deciding what features would be useful to use as predictors is known as feature engineering.

A loss function is used to evaluate how well a model is performing, with the value returned from the function being referred to as the loss. Binary classification problems will often use a loss function known as cross-entropy loss.

Examples of important supervised learning algorithms include linear regression, logistic regression, support vector machines (SVMs), random forests, and neural networks. Artificial Neural Networks (ANNs) are modeled after the human brain and how biological neurons may operate, though their modern-day implementation is far different. The most basic unit within an ANN is the perceptron. A perceptron multiplies each input by a weight and adds them together to produce a weighted sum, after which a step function is applied. A perception is pictured in Figure 7.

![Perceptron Diagram](image)

Figure 7. A modern perceptron, as originally conceived by Rosenblatt (1958).

Stacking multiple perceptrons together creates a network capable of solving problems. A Multilayer Perceptron (MLP) is formed when several layers of stacked perceptrons are used together, including an input layer, one or more hidden layers, and an output layer. In a classification MLP, a softmax function is often used to ensure probabilities all lie between 0 and 1, and that the probabilities for each class add up to 1 (Géron, 2019). When each neuron
in a layer is connected to all other neurons in an adjacent layer, it can be referred to as a fully-connected layer (Géron, 2019). A simple MLP is pictured in Figure 8.

![Figure 8. A simple MLP neural network used for classification.](image)

An MLP is the most basic type of neural network. Deep learning, or deep neural networks (DNNs), refers to neural networks containing large numbers of layers. They often excel at tasks requiring high dimensional inputs, such as image classification, speech recognition, or natural language processing. DNNs therefore don’t always require a feature engineering phase, as some neural networks are designed to directly handle raw input such as images. These are often referred to as end-to-end or featureless models. Deep learning has become extremely popular in the last decade due to advances in compute power, graphical processing units (GPUs), algorithmic improvements, and the widespread availability of larger quantities of data.

High dimensional inputs such as images often make it impractical to fully connect each input. A more desirable and less computationally expensive approach is to connect local regions of the input instead. This approach is typically achieved using Convolutional Neural Networks. A CNN typically includes convolutional layers, pooling layers, and fully connected layers (Stanford University, n.d.).

Convolutional layers are used to learn higher-level feature representations about the raw input data. They achieve this by sliding a small filter across the raw input. The filter (or
kernel) is merely a small collection of learned weights. Convolution is applied by computing dot products of the kernel entries and the input in each region, producing an activation map (or feature map).

A pooling layer then applies a downsampling operation to reduce the size of the activation map, largely to limit the number of connections required within the fully connected layers, thereby decreasing computational costs. A max pooling layer is the most common, where a maximum value is calculated for each patch within the feature maps. A stride and filter size are the two required hyperparameters used to define the degree to which scaling will occur. An example of a max pooling layer can be found in Figure 9. Likewise, an average pooling layer calculates an average value for each patch. An $L_2$ norm pooling layer is another less common option.

![Figure 9. Demonstration of a max pooling layer.](image)

Asokan (2020) explains that another beneficial byproduct of pooling layers is the introduction of local translation invariance to the network. Translation invariance is a property describing the detection of a feature regardless of its location within the input. Asokan further explains that convolutional filters learn local features for a given input without considering them in a global context.

After the size of the data representation is reduced by the pooling layer, fully connected layers then create connections between each of the activations, using matrix
multiplication to generate the final class scores. These layers behave much like the standard ANNs described earlier in the section.

Recurrent Neural Networks (RNNs) are another form of neural network ideal for use with sequence prediction problems, as CNNs are unable to handle variable length sequence data. RNNs are typically used for applications such as handwriting recognition and speech recognition. They operate by maintaining an internal state that is updated at each timestep. The use of RNNs was briefly considered by the authors of MalConv but will not play a central role in this dissertation (Raff et al., 2018a).

Adversarial Examples

Szegedy et al. (2013) first demonstrated that neural networks were vulnerable to adversarial examples, describing the neural networks as containing “intrinsic blind spots” and “counter-intuitive properties”. The authors applied an optimization procedure in which small imperceivable changes were introduced to network inputs such that the target model’s prediction error was maximized. The research literature frequently refers to these changes as perturbations.

Goodfellow et al. (2014) later proposed a simple and fast method for generating adversarial examples, which they refer to as the Fast Gradient Sign Method (FGSM). Their algorithm first computes the gradient of the network’s cost function using backpropagation. They then compute the sign of the gradient and multiply it by a small epsilon value to reduce the perceivability of changes. Finally, the authors then add the resulting vector to the original input to yield the adversarial example. The mathematical representation of FGSM is expressed in Equation 1. A visual demonstration can be found in Equation 1 within the introductory chapter of this dissertation.

\[
x' = x + \epsilon \cdot \text{sign}(\Delta_x J(\theta, x, y))
\]


The FGSM is a type of white-box attack, meaning the attacker knows all information about the model, to include its hyperparameters and model weights (Yuan et al., 2019). On the
contrary, a black-box attack involves an attacker with no knowledge of the target network (Yuan et al., 2019). A black-box attack is often the most relevant scenario for malware classification problems, as antivirus products are unlikely to expose details regarding their internal models.

In adaptive black-box attacks the model acts as an oracle, whereby an attacker is allowed to provide inputs to the model and observe the corresponding outputs, very much like a chosen-plaintext attack in the field of cryptography (Chakraborty et al., 2018). This is akin to providing various files to an anti-virus product and then observing if the product classifies them as malicious or benign. The attacker will often use the resulting input-output tuples to train a local surrogate model, where white-box attacks can then be utilized (Chakraborty et al., 2018). Therefore, most adversarial examples are crafted using white-box attacks and can later be transferred to black-box targets when necessary (Yuan et al., 2019). Given that many black-box attack scenarios ultimately utilize white-box attacks behind the scenes, the importance of research exploring white-box attacks within the malware classification domain is surprisingly more relevant than one might initially expect.

In other scenarios it may be possible for the attacker to gain knowledge about the models’ training data distribution, which forms the basis of a non-adaptive black-box attack (Chakraborty et al., 2018). Just as in the previous scenario, the attacker uses this information to train a local model that allows for the use of white-box attacks (Chakraborty et al., 2018).

Adversarial examples can have varying attack goals, to include confidence reduction, non-targeted misclassifications, or targeted misclassifications (Aryal, Gupta, & Abdelsalam, 2021). Targeted attacks attempt to cause models to output a classification of the attacker’s choosing, whereas non-targeted attacks seek to cause any misclassification to occur without regard to a specific outcome (Yuan et al., 2019). The relative difficulty of attack goals and capabilities can be seen in Figure 10.
With binary classification problems it can be observed that targeted and non-targeted attacks are equivalent (Yuan et al., 2019). When generating an adversarial example using a malicious executable, for example, the adversary can either directly target the benign classification label, or alternatively seek to cause a non-targeted misclassification to occur with respect to the malicious class. Following the successful completion of either attack, a benign classification is output by the model. This demonstrates that targeted and non-targeted attacks produce identical outcomes when applied to binary classification tasks.

Kurakin, Goodfellow, & Bengio (2016) later adapted FGSM to work with targeted attacks in an approach called One-Step Target Class Method (OTCM). The modified algorithm works by using a target label instead of the actual label, and then subtracting the resulting perturbation from the input instead of adding it, as shown in Equation 2 (Kurakin et al., 2016). The difference in implementation amounts to minimizing the loss with respect to a target classification instead of maximizing the loss with respect to the actual classification.

\[
x' = x - \epsilon \cdot \text{sign}(\Delta_x J(\theta, x, y_{\text{target}}))
\]

Equation 2. One-Step Target Class Method.

Kurakin et al. (2016) also proposed an iterative version of FGSM that they call the Basic Iterative Method (BIM). BIM applies FGSM multiple times using small step sizes, clipping the perturbation at each iteration to prevent large changes from occurring (Kurakin et al. 2016). The authors also apply an iterative approach the targeted version of FGSM, which
also uses small changes that are clipped at each iteration. The authors call this algorithm the Iterative Least-Likely Class Method (ILLC). After the introduction of iterative attacks, FGSM and OTCM became known as one-time attacks in the research literature (Yuan et al., 2019).

Each of the previously described algorithms can be used by the attackers with different goals. Adversarial examples can aim to generate either false negatives or false positives (Yuan et al., 2019). A false positive is a type 1 error in which a negative sample is classified with a positive label, and a false negative is a type 2 error in which a positive sample is classified with a negative label (Yuan et al., 2019). Using malware classification as an example, a false positive occurs when an attacker gets a benign sample to misclassify as malicious, and a false negative occurs when an attacker gets a malicious sample to misclassify as benign (Yuan et al., 2019).

**Malware Detection**

Antivirus products traditionally used signature-based methods for detecting malicious files, but an inability to detect previously unseen malware samples led to the rise of behavioral-based detection methods (Aryal et al., 2021). Modern day antivirus products now employ a hybrid approach, utilizing both signature- and behavioral-based methods (Aryal et al., 2021).

According to Malanov (2016), a signature is a contiguous sequence of bytes used to uniquely identify a malware sample. In common usage the term is used more broadly to describe any entry within an antivirus database used to identify a known piece of malware (Malanov, 2016). Sikorski & Honig (2012) explain a common method used to identify these unique sequences of bytes within programs, whereby human analysts can use a utility known as *strings* to search a binary program for human-readable text. The text appears as a series of ASCII or Unicode characters followed by a NULL terminator (Sikorski & Honig, 2012).

Sikorski & Honig (2012) also describe another common signature found in malware databases known as a cryptographic hash. They demonstrate that hashing algorithms such as Message-Digest Algorithm 5 (MD5) and Secure Hash Algorithm 1 (SHA1) can be used to uniquely identify a file, and that searching for the resulting hash in an online malware database can determine if the file has been flagged by an antivirus vendor as malicious. Each
antivirus vendor will have different signatures in their database, making search results for a cryptographic hash vary between each product (Sikorski & Honig, 2012).

By the year 2018, an astounding 376,639 new malware samples were being created every single day, or roughly 4.4 per second (AV-TEST, 2020). Gibert, Mateu, & Planes (2020) attribute the large proliferation of malware samples in part due to the use of polymorphic and metamorphic malware. The authors show that signature-based detection techniques have quickly lost effectiveness, unable to keep pace with the rapid increase in new malware variants. Signature-based approaches lack scalability due to the upfront analysis required by a human expert (Gibert, Mateu, & Planes, 2020).

Despite obvious shortcomings in the ability to detect zero-day malware, signature-based methods are still employed within modern anti-virus products as part of a hybrid approach to detection (Aryal et al., 2021). Key advantages of signature-based methods are their fast lookup speeds and extremely low false positive rates (Aryal et al., 2021). By consulting a hash database as the first step in a hybrid detection pipeline, a positive match within the database would allow computationally expensive steps to be skipped, thereby improving the overall speed and efficiency of the detection system (Chen et al., 2019).

Schultz, Eskin, Zadok, & Stolfo (2000) were the first to propose using machine learning algorithms to classify executable programs. The authors successfully demonstrated that their method could detect previously unseen malicious executables, achieving a 97.76% detection rate using the Multi-Naive Bayes algorithm. The authors reported that their machine learning-based detection method was able to identify over twice as many malicious executables as antivirus products using traditional methods.

The input to machine learning models is typically generated using feature engineering, where an expert identifies important properties that should be extracted from each sample (Ling et al., 2021). With respect to Windows PE malware detection, features can be extracted either statically, dynamically, or even in a hybrid fashion (Ling et al., 2021). Static features are gathered without running the executable, whereas dynamic features rely on monitoring runtime characteristics in a sandbox environment (Ling et al., 2021).

Examples of static features include readable strings, header information, opcodes, API imports, and even byte sequences (Ling et al., 2021). Examples of features gathered dynamically include system calls, CPU utilization, and network calls (Ling et al., 2021).
Figure 11 demonstrates how engineered features are used as the input to a variety of machine learning models.

![Figure 11. ML malware detection workflow, adapted from Ling et al. (2021).](image)

N-grams are a popular feature used within malware detection models, often used with either raw bytes or assembly instructions (Raff et al., 2018b). N-grams can be generated statically and with little domain knowledge necessary, as they work by extracting every unique combination of $n$ bytes or $n$ assembly instructions (Raff et al., 2018b). Their relative ease of use and ability to achieve accuracies above 95% make them an extremely attractive option for malware detection models (Raff et al., 2018b).

**Deep Learning for Malware Detection**

The use of deep neural networks for malware detection appears quite frequently in the research literature throughout the past decade, with earlier approaches focusing primarily on models requiring up-front feature engineering. Motivation for end-to-end raw byte models stems from the desire to eliminate the need for domain expertise and manual feature selection, and to take advantage of powerful and readily available deep learning APIs (Anderson, 2017). Libraries such as Tensorflow, Theano, Keras, Caffe, and Torch have made state of the art deep learning architectures readily available to the masses with only a few lines of code (Anderson, 2017). Deep learning techniques driven by these libraries were dominating leaderboards for problems within other domains (Anderson, 2017).

At the Black Hat USA 2015, Davis & Wolff (2015) built a deep CNN using disassembled instructions from Windows PE files and then briefly discussed the challenges
associated with using raw-byte classifiers. The disassembled instructions allowed for much smaller inputs than using the full raw bytes of a file, making it a much more feasible approach at the time (Wolff & Davis, 2015).

Saxe & Berlin (2015) demonstrated the feasibility of building and deploying a deep learning-based malware classifier capable of delivering fast results with high accuracy. Their deep neural network was trained on 400,000 binaries and resulted in a 95% detection rate at a 0.1% FPR. The model uses a feature engineering approach in which several of the features are extracted using traditional methods such as PE imports and PE metadata. A unique addition to the input features includes a histogram of byte entropy values obtained by sliding a 1024-byte window across the input binary in step increments of 256 bytes, computing an entropy value for each window (Saxe & Berlin, 2015). This unique approach provides some context as to the file’s byte values without the need to keep large amounts of binary content in memory, thus allowing for a small, fixed length feature vector that greatly reduces the memory and CPU constraints necessary for training and loading the model (Saxe & Berlin, 2015).

Anderson (2017) later proposed MalwaResNet, a semi-successful prototype of an end-to-end raw byte model. MalwaResNet is an adaption of the popular ResNet architecture, using a 34-layer deep neural network to determine the maliciousness of a file (Anderson, 2017). The classifier splits input files into multiple chunks to allow them to fit within memory in most GPUs (Anderson, 2017). The results were unimpressive, with models taking days to train and still only yielding an 84.7% true positive rate (Anderson, 2017).

Raff et al. (2018a) later proposed MalConv, a CNN-based raw-byte malware classifier designed to overcome many of the complexity and accuracy issues present in previous end-to-end deep learning models. MalConv is a relatively shallow network capable of processing raw byte sequences of over two million bytes (Raff et al., 2018a). An illustration of the MalConv architecture can be found in Figure 12.
New inputs into MalConv will first undergo a tokenization step, where each byte value is shifted up by one such that byte 0x00 corresponds to a token value of 1 (Raff et al., 2018a). The token value of 0 is then used to represent padding at the end of the file (Raff et al., 2018a). The architecture then converts each token to an 8-dimensional vector using a trainable embedding layer, preventing the model from interpreting byte values as if they were intrinsically closer to other byte values (Raff et al., 2018a).

To best provide location invariance, a convolutional network architecture was selected for the next phase of the model, allowing for detection of malicious features even if they appear at different locations amongst various malware samples (Raff et al., 2018a). By combining convolutional activations with a max-pooling layer, the activations will be produced regardless of where the features are located within the input file (Raff et al., 2018a). Max-pooling was selected over average-pooling because malicious features may only occur once within a file (Raff et al., 2018a). The use of average-pooling would prevent parts of the file with high activations from being nullified by smaller activations that occur at much higher frequencies throughout the remainder of the file (Raff et al., 2018a). Large convolutional filters of width 500 and an aggressive stride length of 500 were selected as to limit GPU memory consumption during backpropagation and improve the balance of computational workloads (Raff et al., 2018a).

Raff et al. (2018a) demonstrated the efficiency and accuracy of the MalConv architecture by testing on a dataset with over 2 million binaries, of which roughly half of the samples were malicious and half were benign. Results show that MalConv continues to improve with larger training sets and provides a strong ability to generalize, whereas prior
byte n-gram approaches would tend to plateau in part due to susceptibility to overfitting (Raff et al., 2018a). When trained on the large dataset of over 2 million binaries, MalConv showed an accuracy of 94.0 and AUC of 98.1 on one test set, and an accuracy of 90.9 and AUC of 98.2 on another test set (Raff et al., 2018a).

Anderson & Roth (2018) curated the EMBER dataset of over 1.1 million binaries and then used it to train another version of MalConv. Due to memory constraints when using two Titan X (Pascal) GPUs, the authors elected to use a batch size of 100 rather than 256, and 1 MB binary sizes instead of 2 MB (Anderson & Roth, 2018). The pre-trained weight files were then publicly released on Github for use within the research community (Anderson & Roth, 2018).

Krcál, Švec, Bálek, & Jasek (2018) independently developed an end-to-end CNN-based malware classifier with similar properties as MalConv. Both architectures use an 8-dimensional embedding layer followed by convolutional layers. By contrast, the architecture proposed by Krcál et al. (2018) is much deeper, contains smaller kernel sizes and stride lengths, and uses a global average pooling layer instead of a global max pooling layer. Krcál et al. (2018) trained and evaluated their model using a much larger dataset of 20 million unpacked Windows PE files, provided by Avast. The authors report slightly better performance than MalConv, with higher accuracy and AUC scores. Other researchers would later begin referring to this model as AvastConv (Raff et al., 2020). A comparison of the two architectures can be seen in Figure 13.
Fleshman, Raff, Zak, McLean, & Nicholas (2018b) conducted a study comparing machine learning based malware detection systems such as MalConv and byte n-gram models against four top commercial anti-virus products, evaluating their robustness after applying a series of four non-adversarial attacks. The attacks include applying benign modifications, destructive byte-level manipulations, injecting return-oriented programming instructions, and packing the binaries. The authors reported that the two machine learning systems were far more successful at detecting malware when evasive techniques were applied. They conclude that machine learning systems are far better at generalizing malicious behavior. The traditional AV products had rigid decision boundaries and were easily fooled by the simplest of modifications (Fleshman et al., 2018b).
Practical Adversarial Attacks Against MalConv

Adversarial attacks using Windows PE files used to be very theoretical in nature, where many papers within the research literature would propose outputting representative feature vectors instead of fully runnable programs (Park & Yener, 2020). Researchers even considered simpler goals, such as preserving the correct format of a Windows PE file, deferring preservation of functionality to future research (Ling et al., 2021). As shown in Figure 14, preservation of maliciousness requires preservation of functionality, which in turn requires preservation of the file format (Ling et al., 2021). Simpler goals could also include a reduction in confidence levels if unable to achieve a misclassification (Aryal et al., 2021).

Figure 14. Relationship between preservation goals, inspired by Ling et al. (2021).

Kolosnjaji et al. (2018) first described an attack against MalConv in which bytes are appended to the end of a target PE file to ensure that the file’s functionality remains intact. A primary challenge the authors must overcome is MalConv’s non-differentiable embedding layer. The authors solve this by embedding each padding byte, computing a gradient with respect to the embedded representation of the byte, and then mapping backwards to input space by finding the closest byte value (0-255) to the gradient-adjusted embedded representation. The formal attack algorithm proposed by Kolosnjaji et al. (2018) can be seen in Figure 15.
Kolosnjaji et al. (2018) compared their proposed gradient-based attack to a random byte attack. The authors report that the gradient-based attack performed significantly better, achieving an evasion rate of over 60% after appending 10,000 bytes to the file, where success rates appeared to grow linearly based on the number of appended bytes. The distribution of byte values within the final payload was expectedly uniform for the randomized byte attack, but the gradient-based attack resulted in an extremely sparse set of byte values (Kolosnjaji et al., 2018). A notable limitation to the attack is payload sizes having to fit within the remaining padding space between the end of the file and the input model’s maximum length (Kolosnjaji et al., 2018). Files that are larger than the 2 MB input size allowed by MalConv must be truncated, making this attack impossible since a payload appended to the end of the file would be removed prior to its consumption into the detection model.

Kreuk et al. (2018) show that payloads can be re-positioned to other spots within the executable without losing their effectiveness. The authors also demonstrate that working payloads can be transferred to different executables altogether with a 75% success rate.

The attack by Kolosnjaji et al. (2018) perturbs each payload byte one at a time. Kreuk et al. (2018) proposed a new attack utilizing the Fast Gradient Sign Method (FGSM, thereby
perturbing all payload bytes at the same time. The attack runs bytes through the embedding layer to obtain an embedded representation of each byte, iteratively runs FGSM on the embedded bytes until evasion is achieved, and then attempts to map backwards through the embedding layer by locating the closest possible byte (Kruek et al., 2019). The improved algorithm achieved an approximate 99% evasion rate with payload lengths greatly decreasing to lengths between 500 and 1000 bytes (Kreuk et al., 2018). The attack algorithm proposed by Kreuk et al. (2018) can be seen in Figure 16.

As mentioned earlier in this chapter, MalConv uses convolutional activations and a temporal-max-pooling layer to produce activations regardless of where features appear within a given input file (Raff et al., 2018a). Kreuk et al. (2018) explain that adversarial attacks against MalConv work by producing large activations that mimic highly benign features, thereby acting as a distraction for the malicious code. Suciu et al. (2019) further explains that MalConv only uses the best 128 activations, resulting in legitimate features being quickly replaced by those from the appended payload in the temporal-max-pooling operation.

Observing that the convergence time of gradient-based attacks appears to grow linearly based on payload size, Suciu et al. (2019) propose a non-iterative, “one shot” Fast Gradient Method (FGM). Results show that success rates increase dramatically based on the quality and robustness of the trained model. Using a payload size of 10,000 bytes with the

```
Algorithm FGSM overlay attack

Input: Original bytes $x$, target label $y$, conv size $c$, embedding matrix $M$

1: $k \leftarrow c + (c - \text{len}(x)) \mod c$
2: $x_{\text{payload}} \sim U(0, 1)^k$
3: $z_{\text{payload}} \leftarrow M(x_{\text{payload}})$
4: $z \leftarrow M(x)$
5: $z^{\text{new}} \leftarrow [z; z_{\text{payload}}]$
6: while $g(y)(z^{\text{new}}) \geq 0.5$ do
7:     $z^{\text{new}} \leftarrow z^{\text{payload}} - \epsilon \cdot \text{sign}((z_{\text{new}} - y))$
8: end while
9: for $i \leftarrow 0$ to $\text{len}(x)$ do
10:     $x_i^{\text{new}} \leftarrow \arg \min_j d(x_i^{\text{new}}, M_j)$
11: end for
12: return $x^{\text{new}}$
```

Figure 16. MalConv attack algorithm proposed by Kreuk et al. (2018).
newly proposed one-shot FGM attack, the authors achieved a success rate of 1% on a model trained with a small dataset, 33% on the EMBER-trained MalConv model, and 71% on a model trained with a large production-quality dataset (Suciu et al., 2019). The small dataset contained less than 10,000 samples and was intended to mimic the size and distribution of the dataset used by Kolosnjaji et al. (2018). The large dataset contained 16.3 million binaries and was produced by using stratified sampling on a larger dataset of over 33 million binaries (Suciu et al., 2019). The use of stratified sampling limited overrepresented of popular malware families, ensured less bias was present, and led to a more uniform distribution of various binary types (Suciu et al., 2019).

Demetrio et al. (2019) demonstrated that MalConv can be evaded by modifying only a select few bytes within the DOS header. The authors then use feature attribution to show that MalConv does not learn meaningful features from the data or text sections of executables, but rather tends to make decisions based on activations produced from within the PE header.

Demetrio et al. (2021b) later extend upon their previous research by introducing three additional Windows PE modification techniques. The authors include these attacks in RAMEN, their newly proposed framework for creating and optimizing practical adversarial malware. The new PE modification techniques are named Full DOS, Extend, and Shift. The Full DOS attack makes use of the entire DOS header except for the leading “magic byte” field and the e_lfanew pointer (Demetrio et al., 2021b). The Extend attack attempts to enlarge the DOS header by adjusting various fields such as the offset to the PE header (Demetrio et al., 2021b). The Shift attack will move the content of the first section forward to create more space, requiring values from various header fields to be adjusted (Demetrio et al., 2021b). The authors released implementations of each attack in an open-source Github repository called SecML Malware.

Adversarial attacks against malware classifiers are not limited to gradient-based attacks. Anderson et al. (2018) demonstrated the use of reinforcement learning to attack models in a black-box scenario. The authors believe the attack more closely mimics a real-life scenario in which attackers will not have access to information about the internal machine learning models used by anti-virus vendors. A reinforcement learning agent makes modifications to the target binary using an action space consisting of various functionality preserving options such as altering section names or creating new but unused sections.
(Anderson et al., 2018). Although not specifically mentioned within the paper, the authors provide a link to an open-source Github repository for the project, where MalConv is available as a target model.

**Adversarial Defenses for PE Files**

Defenses against adversarial malware samples is an underdeveloped area of research, especially as it pertains to end-to-end models. According to a recent survey paper by Park & Yener (2020), it remains unclear if many defenses proposed for use with adversarial images will work within the adversarial malware domain. The authors specifically cite smoothing and randomization as ideas for future research directions within the malware classification domain.

Fleshman, Raff, Sylvester, Forsyth, & McLean (2018a) propose a modified version of MalConv that appears to offer one of the strongest defenses against adversarial attacks. The authors suggest using non-negative weights only, in which classifiers only learn features associated with the positive (malicious) class. The goal is that attackers would not be able to add adversarial bytes to a file to make it appear more benign, as they would fail to produce a strong activation within MalConv. This forces the attacker to instead remove malicious features as a means of lowering the maliciousness score, which is itself a desirable outcome from a defense perspective. The authors reported that non-negative MalConv models prevented nearly all adversarial attacks from occurring. One drawback is that the authors reported lower detection rates, in which the accuracy dropped from 94.1% with the original MalConv model to 89.4% with the non-negative MalConv model.

The most common defense used by malware detection models is adversarial training. Adversarial training is the process of augmenting training data with adversarial examples to produce a hardened and robust network capable of defending against adversarial attacks (Shafahi et al., 2019). Despite its success with most applications, adversarial training incurs an extremely high computational cost, often taking 3-30 times longer than the normal training process (Shafahi et al., 2019). This is considered an unacceptable cost in some instances, as the normal training process can be extremely expensive on its own. Consider, for example, that a popular pre-trained MalConv model released with the EMBER dataset had been trained using dual Titan X (Pascal) GPUs for 10 days (Anderson & Roth, 2018). Performance
considerations aside, Chen et al. (2019) showed that the introduction of adversarial examples into the training process would significantly reduce the likelihood of a successful evasion attack against MalConv. Unfortunately, their research also showed that the introduction of large amounts of adversarial examples into the training process also resulted in malware samples being incorrectly classified as benign after having previously been classified correctly without the use of adversarial training. Moreover, Demetrio et al. (2021b) state that adversarial training is and will likely remain an ineffective technique due to the high dimensionality of input space required within the malware domain.

Taking model defense in a different direction, a survey paper by Park & Yener (2020) suggested that future research should attempt to use pre-processing steps and detection pipelines as opposed to relying on the machine learning models alone. Research has indeed started to move in that direction. Chen et al. (2019) proposed a pre-detection mechanism to reject adversarial examples before they enter the MalConv classifier. In their proposal, the authors suggest creating a hash of both the code section of the executable and the full file itself, and then storing both of those hashes in a database. They reason that if a future input contains an identical code section to a previously identified piece of malware, but the hashes of the two files are different, then someone has likely tampered with that previous malware sample to evade detection. An illustration of this architecture can be found in Figure 17. The authors report that their experiment successfully rejected 100% of adversarial examples that they attempted to use. Furthermore, while detection pipelines often lead to increased processing times, the authors demonstrated that CPU usage was minimal and could even make the pipeline more efficient than using MalConv alone, as finding known malware in the database through the hash lookup process would allow you to skip the MalConv model altogether. They found that this efficiency increase occurred when malware exceeded 0.26% of the input files being tested. Finally, the authors conclude by noting that the introduction of a MySQL database could introduce vulnerabilities of its own, and future adversarial attacks utilizing the code section of the executable could be much more difficult to detect. Finally, note that the pre-detection mechanism may have been inspired by SafetyNet, an architecture which attempts to detect adversarial examples before entering the machine learning model (Lu, Issaranon, & Forsyth, 2017).
Figure 17. The pre-detection mechanism, adapted from Chen et al. (2019).
CHAPTER 3

RESEARCH METHODOLOGY

This chapter will propose several novel techniques for improving adversarial attacks against MalConv. A design science research methodology will be used, where several new technical innovations will be presented and then validated, largely using statistical difference-making experiments. Attacks described in previous studies will first be implemented and included in a new attack tool called BitCamo, with the original attacks serving as a baseline for comparison to the proposed improvements. The new attack tool will also assist in growing the limited number of open-source tools available to create adversarial examples using Windows PE files. Specific implementation details for this research artifact and its proposed improvements will be discussed in great depth throughout in this chapter. This chapter concludes by providing discussion about how the newly proposed treatments will be validated, to include careful attention to dataset selection.

Design Science

The attacks proposed in this chapter are technical innovations requiring a problem-solving paradigm, which describe a design science research methodology (Hevner, March, Park, & Ram, 2014). Wieringa (2014, p. 3) defines design science as “the design and investigation of artifacts in context”. He outlines a repeatable problem-solving process for conducting design science research, which he calls the engineering cycle. He outlines five major stages within this cycle:

- Problem investigation
- Treatment Design
- Treatment Validation
- Treatment Implementation
- Treatment Evaluation
Design science research begin with a design problem, which Wieringa (2014, p. 15) defines as “a problem to (re)design an artifact so that it better contributes to the achievement of some goal”. Artifacts include items such as software, algorithms, or frameworks, and can be more broadly defined as anything developed by someone to serve a practical purpose (Wieringa, 2014, p. 29). Similarly, a treatment can be defined as the way in which these artifacts interact with their problem contexts (Wieringa, 2014, p. 28). Treatment validation ensures that the treatment will support stakeholder goals if implemented and is generally performed in a laboratory setting (Wieringa, 2014, p. 31). Treatment evaluation will assess if the treatment is successful when used by stakeholders in a real-world setting (Wieringa, 2014, p. 31).

The design problems being addressed are low or inconsistent success rates for attacks, high computational costs for attacks, and the inability for attacks to work when a pre-detection mechanism is employed. The attack tool is the artifact, and when the attack tool is run against executables with the new improvements enabled, it will be referred to as the treatment. The population is a diverse collection of over 10 million malicious Windows PE files intended to be representative of all Windows PE files that one may expect to find in a real-world setting. To perform treatment evaluation, the artifact will be applied to a subset of samples selected uniformly at random from this population.

Dataset selection will be addressed in a section later in this chapter. Treatment implementation and design will be discussed throughout this chapter. Treatment validation is provided with the results presented in chapter 4.

**Statistical Difference-Making Experiments**

Within the context of design science, many of the experiments performed for this research can be classified as statistical difference-making experiments. Wieringa (2014, pp. 47-48) explains that statistical difference-making experiments apply treatments to different subsets of samples drawn from a shared population and then seek to determine if a difference exists in the average outcomes. In the context of this research, successful outcomes are determined by the presence of high evasion rates, small average payload sizes, and small average attack durations.
Stakeholders

Stakeholders are the people affected by the treatment of a problem (Wieringa, 2014). According to Wieringa, a stakeholder need not be aware of the existence of a problem nor the associated treatment, but they must be better off once the problem is treated. It can be argued that the primary stakeholders and beneficiaries to all malware-related research are therefore the end users of computer systems. Published malware research, even when offensively oriented, is consumed by companies offering antivirus products and security solutions, and in-turn used to protect end users from malicious software. The open-source library is most likely to be consumed directly by security researchers, penetration testers, red team operators, security architects, and machine learning engineers.

Research Questions and Hypotheses

Research Question 1
Can the append-based FGSM attack be re-implemented and improved to achieve consistently high evasion rates?

Hypothesis 1
Success rates vary widely within the research literature. Dataset and implementation differences likely play a key role. The attack can likely be re-implemented to achieve moderately high success rates with payloads of 1000 bytes or less. As payload sizes increase to several thousand bytes, success rates will likely surpass 99%.

Research Question 2
Can the append-based FGSM attack benefit from a new byte initialization strategy?

Hypothesis 2
Some researchers consider the random byte initialization strategy to be naïve. It likely performs better than methods that initialize every byte with the same value, but improvements are likely possible by drawing from a pool of values that are more representative with respect to the target classification.
**Research Question 3**
Can the recently proposed pre-detection mechanism be completely bypassed?

**Hypothesis 3**
The pre-detection mechanism can be trivially bypassed with almost every sample.

**Research Question 4**
Can performance of the long-running reconstruction phase of the append-based FGSM attack be improved?

**Hypothesis 4**
The reconstruction phase is likely to remain the bottleneck for the attack, but some simple improvements are likely to make a big difference.

**Novel Contributions and Proposed Attack Improvements**

An attack tool will be developed and released via open-source website Github for use within the research community. The lack of readily available and easy-to-use attack software may contribute to a lack of urgency in developing proper defenses for adversarial attacks. The lack of publicly available attack software also increases the difficulty to improve upon existing attacks and discourages continued research into this area. The tool will make every effort to automatically determine optimal parameters to run the attack with, allowing users to run the tool without any up-front configuration or need to understand which attack works best for a given situation.

New payload initialization strategies will be proposed and then evaluated. Previous researchers indicate that the use of random payload initializations may be a less-than-optimal strategy (Chen et al., 2019). Simple strategies were attempted first, such as filling the entire payload with the exact same byte value. Initial attempts exclusively targeted outlier values and midpoints, to include 0, 1, 128, or 255. After the successful implementation of performance improvements, this strategy was amended to include all 256 possible byte
values, reiterating the importance of the engineering cycle described by Wieringa (2014).

While conducting the research, a similar strategy was discovered in a blog post written by Fleshman (2018a), a co-author of the original MalConv proposal. Fleshman found optimal results when using byte value 169 (0xA9). The approach used in this dissertation differs in the following ways:

- The payload bytes are used as a basis for a gradient-based algorithm instead of merely appending them to the file without modification. The byte value used to initialize the payload will not necessarily be reflective of the byte values contained within the final payload. The results reported in chapter 4 will reinforce this distinction, as the optimal byte identified by Fleshman results in low evasion rates after being perturbed by FGSM.

- Results are gathered over a substantially larger dataset. Fleshman uses 50 executables included as part of a competition dataset, whereas this dissertation tests all 256 possible byte initialization strategies against 1,000 executables. The top eight candidates are later tested against an even larger dataset of 5,000 executables to provide validation of the initial results.

- Fleshman achieves evasion by adding a new section containing 100,000 bytes. This research will test with using much smaller payloads via use of 1,000-byte file overlays.

A weighted initialization scheme is also proposed. The intention is to maintain a data structure containing byte distribution counts from previously successful payloads. Whenever an attack succeeds, for example, the tool will count how many occurrences of each possible byte value exist within the payload. Subsequent successful runs will increment counts within this same data structure. This results in a count for all possible byte values. The weighted strategy uses this probability distribution to initialize new payloads. Bytes occurring at high frequencies within successful payloads will therefore appear at high frequencies within newly initiated payloads. The motivation for this strategy is to give the algorithm a head start, initiating payloads with values that already resemble successful payloads.

The byte reconstruction phase of the attack appears to be a significant bottleneck in terms of computation time. Recall that after running a gradient attack against the embedded representation of the bytes, the embeddings need to be mapped backwards across the
embedding layer to a byte value in the range 0-255. This dissertation presents three improvements to improve the speed of the reconstruction phase:

- Applying reconstruction to the payload bytes only.
- Adding parallelism.
- Replacing the L_2 distance norm with K-D tree queries.

Psuedocode provided in papers by Kreuk et al. (2018) and Suciu et al. (2019) appear to indicate that the reconstruction phase should loop over all embeddings. It can instead be observed that reconstruction only needs to be applied to the embedded payload, as bytes from the remainder of the file should never change, as this could lead to a loss of functionality. The rest of the file should therefore always retain their original byte values.

The addition of parallelism is an implementation detail that would not necessarily be captured in an algorithm’s pseudocode. Given the significant computational cost associated with this phase of the attack, it was deemed important to verify that this phase can indeed be parallelized and confirm that it offers a significant improvement in processing time.

It was observed that the L_2 distance norm calculations are very much a brute force strategy. In prior attacks, each possible byte value (0-255) is run through the embedding layer, with the resulting 8-D embeddings being stored for future reference. This provides a mapping between all possible byte values and their corresponding embedded representations. This can be performed once at runtime. Later, after completion of the gradient phase of an attack, each embedding within the perturbed payload needs to be converted back to a byte value to allow for creation of a working executable. The perturbed embeddings are unlikely to match one of the 256 embeddings saved from earlier, so the reconstruction phase instead needs to locate the closest embedding from the lookup table. For each perturbed embedding within the payload, an L_2 norm is computed between itself and each of the 256 fixed embeddings to determine which it is closest to. The distance formula is being computed 256 times for each byte within the payload, resulting in an excessive number of square root operations and comparisons.

This dissertation proposes replacing the L_2 distance norm calculations with K-D tree queries. A K-D tree is a space-partitioning data structure utilizing binary trees and is therefore capable of performing searches in logarithmic time on average (Bentley, 1975). A visualization of a K-D tree can be seen in Figure 18. The biggest limitation to K-D trees and other nearest neighbor search problems are their susceptibility to the “curse of
dimensionality”, where performance degrades quickly when using high-dimensional data (Liberty, 2013). On the other hand, K-D trees tend to be very efficient in low-dimensional space, with Python package sklearn advertising that it works best with a dimensionality of 20 or less (Scikit-learn, n.d.). Additionally, Maneewongvatana & Mount (2001) observed that K-D Trees perform well when both the data and the point being queried are of the same dimension. For these reasons, the 8-D embeddings used within the reconstruction phase appeared to be extremely fast.

Figure 18. Visualization of a K-D tree in two dimensions.

The reconstruction phase results in a natural loss in precision, as 8-dimensional vectors of floating-point numbers are converted to single byte values. Gradient attacks against the embedded representation of bytes can therefore succeed but later fail again after being converted back to byte values following the reconstruction phase of the attack. Demetrio, Biggio, Lagorio, Roli, & Armando (2021a) also made this observation, describing that an attack can succeed in feature space but later fail in input space. To overcome this problem, this dissertation proposes the use of excessively strong perturbations within embedding space, such that the perturbed sample will have a greater change of retaining the benign label after completion of the reconstruction phase.

To achieve stronger perturbations, two changes are made to the original attack algorithm proposed by Kreuk et al. (2018). First, instead of stopping the gradient attacks after a MalConv score of 0.50 is reached within embedding space, this dissertation will continue until a score of at least 0.01 is achieved. Second, the need to reduce the strength of the
adversary is not as obvious within the malware domain as it is within the image domain. An excessively strong perturbation is not observable to a human observer in the same way as it is with images. The tool therefore uses high epsilon values to attack each sample.

Some samples achieve evasion using a very small number of adversarial bytes, whereas others require somewhat larger sized payloads. Using fixed sized payloads is therefore suboptimal. The payload size equation provided by Kreuk et al. (2018) allows for more variance in payload sizes but is suboptimal as well. The speed improvements offered in this dissertation would allow for a more aggressive search-based approach, where attacks can be attempted several times with varying payload sizes to determine the smallest payload size yielding a successful result. While a binary search appears suitable for this situation, the observation is made that certain samples can fail with larger payload sizes but succeed with small payload sizes. In these cases, the binary search would result in the payload size increasing even more after a failure, effectively missing the optimal payload size. A new payload size minimization algorithm is proposed, beginning by first running the attack using a 50-byte payload, and then iteratively increasing the payload size in 50-byte increments until the sample finally achieves evasion, up to a user-specified maximum payload size.

Finally, the attack tool will attempt to defeat the pre-detection mechanism by appending a single byte to the code section when slack space is present. Evasion rates will be determined by comparing the original code section hash to the modified code section hash and ensuring a change has occurred.

**Tools and Environment**

Software artifacts for this research were initially developed on a consumer-grade laptop using Windows 11 with Windows Subsystem for Linux (WSL) 2. WSL2 was configured to use Ubuntu 20.04 as its Linux distribution. Visual Studio Code was used as the primary code editor. All source code was checked in to open-source website Github. The Github repository was configured to use a continuous integration pipeline using CircleCI, allowing for automated unit testing after each new code push.

Python 3 was selected as the programming language for the attack tool. Python has one of the largest ecosystems of libraries available for projects related to data science, machine learning, and deep learning. Two of the most popular machine learning libraries used
by the research community are PyTorch and Tensorflow, with PyTorch emerging in recent years as the clear favorite.

Due to the extreme computational cost associated with training a new MalConv model, which tends to require dataset sizes in the millions, an open-source model with pre-trained weights was selected. A model trained using the EMBER dataset was selected for use with this project, created by Anderson & Roth (2018). The pre-trained model is available as a Keras HDF5 file. Instead of applying extra effort to convert this model to something usable by PyTorch, Keras and TF2 were selected for use with this project. Adversarial attack library CleverHans provides support for both PyTorch and Tensorflow, making both frameworks a suitable choice. CleverHans includes many attack algorithms, to include the Fast Gradient Sign Method (Papernot et al., 2016).

Reproducibility of results is important, especially given the differing success levels amongst other research projects investigating attacks against MalConv. Given the selection of Python 3, the use of Anaconda, virtual environments, and a requirements.txt file was used to ensure dependencies are consistent across installations. A Docker image is also provided to allow for maximum portability of the tool and ease of use across different base operating systems.

The Library to Instrument Executable Formats (LIEF) is another Python library selected for use with this research project for its ability to instrument binary executable files, to include the Windows PE file format. Adversarial attacks against Windows PE files call for binary modifications to take place while still preserving program functionality. To that end, a stable and trustworthy library is preferred over manual modifications which may be more susceptible to error.

Implementation of the pre-detection mechanism calls for use of a relational database. Additionally, when running large experiments in bulk, databasing the results provides an easy way to query for specific results afterwards. This proved to be much more helpful than sifting through large log files produced by command line interface (CLI) output from the attack tool. PostgreSQL was selected over alternatives such as MySQL simply due to familiarity of the author. The psycopg2 was used as a database adapter to allow for connectivity between the Python program and the PostgreSQL database.
Running experiments against larger datasets are computationally expensive to complete, as will be discussed in more detail later in the chapter, with some experiments taking several days to run. Experimental results provided in the following chapter will be computed using an Intel i7-8750H CPU, containing 6 cores at 2.20 GHz.

Artifact Design: Overview

An initial goal for the attack tool was to implement the append-based FGSM attack originally described by Kreuk et al. (2018). Only after the original attack was implemented could potential improvements be included and then evaluated. The attack tool is a simple CLI utility written in Python. The tool will be used to support experiments in the next chapter of this dissertation and therefore includes highly customizable command line arguments and the ability to support bulk processing of large datasets.

The only required command line argument for the attack tool is the location of a single executable or directory containing multiple executables. Running the attack tool with the `-h` or `--help` options will display usage instructions describing each of the program arguments. Optional arguments include the ability to customize the output directory, adjust logging verbosity, altering the payload size, allowing existing output executables to be overwritten, changing the payload initialization method, saving experiment results to a database, or to invert behavior by using benign input files.

Type 2 errors are of the most interest to an attacker, where they seek to modify a malicious executable such that a machine learning model classifies it as benign. For the purposes of early development on the attack tool, it was easier to work with benign executables, allowing for continued use of an anti-virus product in the development environment. An observation rarely mentioned in the research literature is that most of the proposed attacks can work in either direction, allowing benign executables to be misclassified as being malicious while remaining completely safe to run. Malicious executables were tested later in the development process. The command line flags `-b` or `--benign` are therefore used to tell the tool the input files are benign, and the targeted output classification should be malicious.

An input file or directory is a required argument. For ease of use, the tool does not require the user to specify which is being supplied. The tool will automatically determine if
the argument is a valid file, valid directory, or an invalid input. In the case of a file, validation is also performed to ensure the supplied file is in Windows PE file format. Early in the development of the tool, a simple check for the “MZ” magic bytes was performed. The current version of the tool now uses LIEF to determine if the file is valid. When a directory is supplied as input, the tool will fetch the list of files within the directory and loop through them to identify valid PE files. This functionality allows for bulk processing of executables.

The optional argument for output directory is validated by ensuring the path exists, the path is a directory, and the user has permission to write files within the directory. If an output directory is not specified by the user, modified executables will be saved to the “samples/output” directory.

**Artifact Design: Model Implementation**

To determine success for a given attack, the tool must have the ability to run samples through the complete pre-trained MalConv model. A natural flow for the program consists of:

- Running the original sample through the model.
- Recording the prediction for the original sample.
- Running an attack to modify the original sample.
- Running the modified sample through the model.
- Recording the prediction for the modified sample.
- Comparing results and determining if evasion has been achieved.

Adding the ability to run samples through MalConv was therefore the next step taken in development of the attack tool. White-box attacks will need the ability to compute gradients, also requiring the model to be present first.

As previously mentioned, this study will use the EMBER-trained MalConv model created by Anderson & Roth (2018). The HDF5 file was downloaded from the EMBER Github repository. Keras code used to define the model’s architecture was therefore adapted from code within this repository as well. Code defining the MalConv architecture can be seen in Figure 19.
Figure 19. Keras code defining MalConv, adapted from Anderson & Roth (2018).

This code would not remain intact for long, as the next section of this chapter calls for breaking this model apart to assist with the attack code. With the attack tool now parsing input files and the malware classifier defined, it was then possible to begin running executable files through the tool to determine their maliciousness. This capability was initially tested using a small benign dataset gathered from the Windows development machine itself, and through a small set of malicious executables downloaded from VirusTotal.

Samples smaller than the 1 MB input size used by the EMBER-trained MalConv model were padded using a special character. The EMBER-trained MalConv model uses a value 256 for its special padding character (Anderson & Roth, 2018). This is different than the padding value of 0 specified within an appendix diagram for the original MalConv proposal by Raff et al. (2018a). It was deemed important to use the same convention that the model was trained with. Samples larger than the 1 MB input size were truncated, allowing them to be used with the smaller input size. These executables are not targetable by the attack tool when using an append-based strategy to add a file overlay. This limitation is well documented within the research literature.
Artifact Design: Implementation of Append-Based FGSM Attack

After adding the ability to classify samples using the attack tool, the next step was to implement the append-based FGSM attack proposed by Kreuk et al., (2018). A primary challenge when attacking MalConv is overcoming the non-differentiable embedding layer. Most gradient-based attacks within the research literature solve this problem by first generating an embedded representation of the input bytes. The Keras layers shown in Figure 19 were therefore broken up into two separate components. The first component takes raw bytes as input, normalizes their size to 1 MB as described in the previous section, and then runs them through the initial embedding layer. The second component uses the embedded 8-dimensional vectors as input and is responsible for running them through the rest of the model. The second component is fully differentiable and can therefore be used to collect gradient information required by the attack code. A key observation is that chaining the two new smaller models together yields equivalent results to running the complete model by itself.

The attack code begins by initiating a user-specified payload size using bytes generated uniformly at random. The payload is appended to the end of the input file. The bytes are then run through the embedding layer only to create an embedded 8-dimensional representation of each byte.

An iterative process then begins by first running the CleverHans implementation of FGSM against the embedded bytes. This iterative process is an implementation of the while loop originally shown in Figure 16. The algorithm appears to call for the perturbed embedded bytes to be subtracted from the original embedded payload only. In practice this appeared to cause a length mismatch error. To overcome this challenge, the perturbation was instead subtracted from the full embedded representation instead of the payload only. After the subtraction is complete, bytes from the payload are extracted and appended to the original embedded representation. This approach ensures that only the payload bytes are ever perturbed. A working implementation of the attack loop is shown in Figure 20.
There are two more subtle differences in which the attack tool implementation differs from the original algorithm shown in Figure 16. The attack tool implementation appears to add the signed gradient instead of subtracting it. This is misleading though, as setting the “targeted” argument to true in the FGSM function call results in CleverHans inverting the loss internally by adding a negative sign to it. Therefore, loss will still be minimized with respect to the target label. The other notable difference from the attack proposed by Kreuk et al. (2018) is the addition of clipping, notably absence in Figure 16. This appeared to provide more consistent results, clipping the embedded bytes to fit within the valid range of values before proceeding to the next iteration. The use of clipping is suggested in the original proposal for an iterative FGSM, commonly referred to as BIM (Kurakin et al., 2016).

The loop will end when the embedded bytes successfully achieve high evasion rates of 99% or greater when ran against the partial MalConv model defined earlier. In most cases this happens quickly, but a maximum iteration count was added for protection against an infinite loop, which does occur when samples are unable to achieve high evasion rates.
The final piece of the attack is mapping backwards across the non-differentiable embedding layer. In other words, a collection of 8-dimensional vectors now exists for which evasion can be achieved. To produce a working executable, each 8-dimensional vector needs to be converted into a byte within the range 0-255. The original attack by Kreuk et al. (2018) explains a method for mapping each embedding to its nearest neighbor within MalConv’s embedding matrix. Suciu et al. (2019) use an L2 distance metric to achieve this. The Python code used to convert a single 8-D embedding into a byte can be seen in Figure 21.

```python
6  def reverse_lookup(embedding_matrix, embedding):
7      
8          Converts a single 8-D embedding to a byte using an L2 distance metric
9          in order to locate the closest possible byte the an embedding. This
10         allows for mapping backwards from embedded space to input space.
11         
12         Implemented as a top-level function to allow for use with parallel
13         processing libraries.
14         
15         Parameters
16         
17         embedding : numpy.ndarray
18             A single 8D embedding array to lookup.
19         
20         Returns
21         
22         byte: np.uint8
23             Embedding space representation of the input byte.
24         
25         distances = [tf.norm(eb - embedding, ord=2) for eb in embedding_matrix]
26         byte Tf = tf.math.argmin(distances)
27         byte_np = np.uint8(np.array(byte_Tf, dtype=np.uint8).item())
28         return byte_np
```

Figure 21. Python function used to find the closest byte to an 8D-embedding.

To convert the entire payload from embedding space to input space involves a loop over each of the payload bytes. This is sometimes referred to as the reconstruction phase. While the original proposal in Figure 16 suggests performing this step for all input bytes within the file, mapping backwards through the embedding layer appears to be a clear bottleneck in the performance, with the attack tool from this paper spending over 99% of its time there. This dissertation observes that this step is only necessary to perform using the payload bytes instead of the full file. The original content needs to remain intact to preserve functionality, making it possible to simply copy them from the original input file. Only the embedded payload must be mapped back to input space. The reconstruction phase can also be
parallelized to improve the runtime of the attack quite substantially. The attack tool uses Python’s Joblib library to implement the parallelism. It uses the tqdm library to display a progress bar for this long-running portion of the attack.

The reconstruction phase often leads to a drop in accuracy. When 8-dimensional vectors of floating-point numbers are converted to single integers within the range 0-255, it’s only natural that multiple vectors would end up mapping to the exact same byte value. These collisions will result in a natural loss in precision, as 20 vectors that were once very different from one another may end up having the same byte value as one another in the final payload.

Multiple hobbyist implementations of this attack on Github make the mistake of using the MalConv prediction from inside the attack loop to declare that evasion has been successfully achieved. After mapping backwards through the embedding layer, the final perturbed binary must be run back through the full model before declaring the attack was successful in evading MalConv. Capturing any loss in precision from the reconstruction phase is vital to ensuring results are accurate.

The results produced from running this baseline attack against a large dataset are presented in chapter 4.

**Artifact Design: Payload Initialization Techniques**

In addition to the randomized payload initialization technique proposed by Kreuk et al. (2018) and Kolosnjaji et al. (2018), this research also adds a pseudorandom initialization method to the attack tool to allow for consistent results, allowing successive runs of the attack tool to produce identical perturbed binaries if desired by the user. After implementing the pseudorandom technique, the command line arguments -i or --initialization-method were added to allow users to overwrite the default behavior and pave the way for experimental payload initialization techniques to be added. The pseudorandom technique was added for convenience and is not expected to offer an improvement to the attack. A source code file to hold all payload initialization functions was also added for organizational purposes.

With the attack tool now supporting multiple payload initialization strategies, it was easy to begin adding new techniques to determine if any of them offered a clear improvement over the randomized method. Implementing techniques that called for the same byte to be
used throughout the entire payload were trivial. Outlier bytes were obvious candidates to try, such as 0 and 255. Other cases such as the midpoint and the padding byte were attempted for completeness. These simple initialization strategies can be seen in Figure 22.

![Code Snippet]

Figure 22. Implementation of simple payload initialization strategies.

More complex payload initialization strategies were also attempted. Of high interest is the weighted initialization strategy, as described earlier in the chapter. This strategy seeks to use the byte distributions from successful payloads as a starting point for future attacks. For example, 0x8B was amongst the bytes that appeared most frequently within payloads that allowed a malicious binary to appear benign. The idea behind the weighted initialization strategy is to use the 0x8B byte at similar rates as it appeared within the payloads of successful attacks, using it as a starting point before running the gradient attack.

To implement the weighted initialization strategy, the attack tool makes use of Python’s `random.choices()` function. When supplied with a `weights` parameter, it allows certain values to be chosen at higher frequencies than others. The full implementation of this initialization strategy can be seen in Figure 23.
To obtain the distributions used by the weighted initialization strategy, 1,000 samples were selected at random and ran through the attack tool using the current randomized initialization strategy. The payloads of each successful attack were saved and then the counts for each byte were tallied. In creating these distribution vectors, care was taken to select different samples than the ones being used for the experiments whose results are reported in chapter 4.

Artifact Design: Defeating the Pre-Detection Mechanism

Chen et al. (2019) proposed a pre-detection mechanism to reject adversarial examples before entering the classifier. The pre-detection mechanism works by computing a hash of both the PE code section and the full file (Chen et al., 2019). When the pre-detection mechanism observes a new sample whose code section is identical to a previous malicious sample but whose full file hash is different, the conclusion can be drawn that the new sample has likely been tampered with and can therefore be immediately rejected as being an adversarial example (Chen et al., 2019). The authors report a 100% success rate but allude to the possibility of code section attacks in future research (Chen et al., 2019).

While Chen et al. (2019) acknowledge that their proposed defense would likely not work against future adversarial attacks that target the code section, complex fine-grained transformations of the code section appear unnecessary to achieve evasion. This dissertation proposes the use of a single byte attack to be carried out against the code section. This is enough to produce a completely different hash of that section. No attempt will be made to move the adversarial payload into the code section. The adversarial bytes will continue to be placed in an area of the PE file that is easier to manipulate, such as the overlay or slack space.
Chen et al. (2019) note that their pre-detection mechanism offers a performance increase when at least 0.26% of new inputs result in a successful database lookup. This occurs because the database lookup is significantly faster than running a new sample through the MalConv model. While the performance increase appears desirable, this likely means the pre-detection mechanism is vulnerable to a timing attack whereby attackers can determine if their sample has been rejected by the pre-detection mechanism or the MalConv model itself. This provides context for where an attacker should focus their attention in subsequent attacks.

It’s often the case that extra space is present at the end of each section, often referred to as slack space. Existing code will not make jumps or calls into this section, making it safe to use without disrupting file functionality. The original proposal for the pre-detection mechanism by Chen et al. (2019) doesn’t specify if the section hash is to be computed on the code section’s content only or if the hash should be taken across the entire section, to include slack space. This dissertation assumes that a valid attack should address both scenarios. Addressing this challenge amounts to modifying header fields such as `virtualSize` so that defenders are unable to differentiate between the original code section instructions and the maliciously appended content.

Previous research has demonstrated that adversarial payloads can also be placed within slack space (Suciu et al., 2019). When specifically applied to the code section’s slack space, this should result in the hash of the code section changing, provided header values are also updated. Slack-space attacks should therefore defeat the pre-detection mechanism. Attacks utilizing the file overlay are naturally able to achieve higher success rates though, as more bytes can typically be placed here (Suciu et al., 2019).

This dissertation therefore continues to place payloads in the file overlay with hopes of achieving optimal success rates and uses only a single byte of slack space within the code section to bypass the pre-detection mechanism. Using all remaining slack space to place an adversarial payload may raise other red flags in a feature-based detection pipeline. All existing slack attacks greedily choose to use all remaining space within a section.

To perform binary patching of the executable and implement the single-byte attack, the attack tool will make use LIEF. The algorithm begins by locating the code section, signified by the `.text` naming convention. The algorithm proceeds by determining if slack space exists by checking to see if a difference exists with respect to the `SizeOfRawData` and
virtualSize fields. If present, the algorithm will conclude by appending a single byte. For demonstration purposes and ease of implementation, the byte 0x01 is appended. In practice it would be advisable to use a random value to prevent defenders from explicitly checking for the byte value 0x01. The full algorithm as implemented in Python can be seen in Figure 24.

```python
1 import array
2 import lief
3 from utils.os import get_code_section, text_section_exists
4
5 def bypass_pre_detection(bytez):
6     patched = False
7     binary = lief.PE.parse(bytes(bytez))
8     section = get_code_section(binary)
9     if section is None:
10         return bytez, False
11
12     gap = section.sizeof_raw_data - section.virtual_size
13     if gap <= 0:
14         return bytez, False
15
16     section.content = section.content + bytes(b"\x01")
17     section.virtual_size = section.virtual_size + 1
18     patched = True
19     builder = lief.PE.Builder(binary)
20     builder.build()
21     return array.array('B', builder.get_build().to_bytes(), patched)
```

Figure 24. Binary patching code used to defeat the pre-detection mechanism.

The artifact was then tested against small subsets of the test datasets to ensure the pre-detection attack was implemented correctly. The resulting binaries were manually analyzed to ensure the LIEF library was patching executables as intended. The expected result was a very minimal set of changes, to include a single additional byte within the code section’s slack space and adjustments to headers that increase the size of the code section.

In nearly all cases the expected result was validated. A select few binaries showed large numbers of modified bytes though. An investigation revealed that LIEF automatically fills slack space with 0x00 byte values, even if the original content used other values. The Windows PE file format should indeed use a 0x00 value within slack space (Goppit, 2006). Despite testing with benign binaries from trustworthy sources, analysis revealed this was not always the case. For example, a Firefox binary had its code section’s slack space filled with 0xCC values. This 0xCC value represents a software breakpoint and will therefore halt code execution. LIEF replaced all 0xCC bytes in this executable’s slack space with 0x00 bytes upon re-writing the binary. This unexpected finding appeared to have no impact on the
success of the pre-detection evasion attack but is documented for the benefit of those wishing to minimize the number of changes that occur to the target binary.

Attacks were then re-run against a subset of the testing dataset to verify that the new changes work as intended. The malicious dataset returned modest results, with only 60.82% of samples evading pre-detection. The results significantly underperformed expectations, leading to an investigation to determine the cause of failure cases. The investigation concluded that several binaries did not have a .text section, preventing binary patching from occurring. Manual analysis of a few binaries exhibiting this behavior revealed obfuscation using various packing tools. For example, several executables had a base code section of UPX1 instead of .text. This likely means the executable was packed using UPX, a free executable packer that offers high compression ratios and fast decompression rates.

Re-examining the proposal for the pre-detection mechanism revealed the authors calling for hashing of the executable section and not the .text section specifically (Chen et al., 2019). The attack implementation was therefore modified to dynamically determine the section containing the beginning of the executable’s code instead of assuming this section will always be named .text. Dynamic resolution was achieved by following the BaseOfCode or AddressOfEntryPoint fields within the Windows PE header. The RVA provided by these fields can be supplied to LIEF’s section_from_rva function to determine the code section. Using the BaseOfCode field did not always result in a successful lookup. In those instances, AddressOfEntryPoint was used as a fallback value. An implementation of the fallback method can be seen in Figure 25.

```python
def get_code_section(binary):
    try:
        # First try to determine section using BaseOfCode
        code_rva = binary.optional_header.baseof_code
        code_section = binary.section_from_rva(code_rva)
        return code_section
    except:
        try:
            # Then try using AddressOfEntryPoint
            code_rva = binary.optional_header.addressof_entrypoint
            code_section = binary.section_from_rva(code_rva)
            return code_section
        except:
            try:
                # Try the standard .text section if those methods don’t work
                code_section = binary.get_section('.text')
                return code_section
            except:
                return None
```

Figure 25. Using LIEF to dynamically determine the code section.
Direct modifications to the code section using semantically equivalent instructions were briefly explored. The disassembly framework Capstone was used to disassemble instructions from the code section. Linearly processing of disassembled instructions appeared problematic, as only the first handful of instructions appeared valid in some instances. The valid instructions often corresponded to a start section, as shown via IDA Pro in Figure 26. Full implementation of this feature would likely entail “stepping into” calls and jumps. A more thorough validation strategy would also be required to ensure program behavior remains intact. Modifying instructions remains a promising research direction for future work.

![IDA Pro view of the valid instructions captured by Capstone.](image)

**Artifact Design: CLI Output**

An example demonstrating usage of the attack tool and its corresponding output will now be provided. For demonstration purposes, a single malicious executable will be perturbed using a 1,500-byte payload, random payload initialization strategy, and $L_2$ distance metric reconstruction method. The attack tool arguments in that scenario may look like:

```
python3 attack.py -p 1500 -i random --l2norm -f samples/malicious/sorel20m/
```

The tool begins by collecting the original file hash and code section hash to determine if the pre-detection mechanism would be evaded. The original maliciousness score determined by MalConv is also provided to the user. In this case, MalConv correctly predicts
the sample is malicious with 100% confidence. In the attack phase, bytes are run through the embedding layer and then iteratively perturbed using FGSM. The maximum number of iterations is configurable, but the attack is designed to stop when MalConv reaches a maliciousness score close to 0. After the embedded bytes are perturbed, the reconstruction stage begins. A progress bar was implemented to provide the user with feedback regarding the reconstruction status. These features can all be seen in Figure 27.

![Figure 27](output-of-the-attack-tool-during-the-reconstruction-phase.png)

After completion of the reconstruction phase, hashes of the final payload are computed for comparison purposes. The single-byte code section attack is then attempted in hopes of evading the pre-detection mechanism. Results are then printed to the screen, including the MalConv score of the final perturbed payload. Scores below the 50% threshold are considered to have achieved evasion. Hashes of the final file and code section are also calculated to determine if the pre-detection mechanism would be evaded. Output printed after the reconstruction phase can be seen in Figure 28.

![Figure 28](output-of-the-attack-tool-post-reconstruction.png)
In the provided example, an executable with a maliciousness score of 100% was perturbed to achieve evasion, with MalConv assigning the modified executable a maliciousness score of 0%. The attack tool will then output cumulative statistics necessary for interpreting results when using the bulk-processing feature is used, to include the percentage of samples able to evade detection by MalConv.

**Dataset Selection**

This section describes the dataset collection methodology to be used throughout the remainder of this dissertation. Due to significant flaws in the methodology used by multiple related papers, a review of existing datasets and their associated shortcomings will first be provided.

Raff et al. (2018b) report heavy bias and overfitting when using datasets containing only a few thousand samples, explaining that they’re not large enough to provide good representation of the full population of executables. Gathering Windows PE files from Microsoft Windows is a common method to obtain benign samples, but the authors explain that they lead to strong bias, where models learn to identify Microsoft compiled executables instead of generalizing to identify any benign file. Likewise, malware sharing websites such as Virus Share tend to contain samples provided by volunteers and human analysts, making them biased towards specific types of malware families (Raff et al., 2018b).

In one of the first adversarial attacks demonstrated against MalConv, Kolosnjaji et al. (2018) produced a small dataset containing 9,195 malicious binaries and 4,000 benign binaries. The malicious files were collected from VirusShare, Citadel, and APT1. The benign files were downloaded at random from popular search engines. Suciu et al. (2019) were unable to reproduce the results of the research reported by Kolosnjaji et al. (2018), and therefore attempted to re-create a dataset of similar size and distribution to see if the results were merely artifacts of the dataset’s properties. They concluded that the dataset suffered from severe overfitting and poor generalizability.

As a result of this finding, Suciu et al. (2019) opted to use a dataset consisting of 16.3 million binaries, which was derived from an even larger dataset of 33 million binaries, collected from VirusTotal, Reversing Labs, and proprietary FireEye data. Stratified sampling was performed to limit the inclusion of overrepresented malware families. They later ran each
experiment using this dataset, the EMBER dataset, and the small overfitted dataset modeled after work performed by Kolosnjaji et al. (2018).

Chen et al. (2019) created a small dataset containing 5,200 malicious binaries and 5,150 benign binaries. The malicious files were collected from VirusShare, DAS, and malwarebenchmark, whereas the benign files were collected from various Microsoft Windows installations and supplemented with executables from 30 additional software companies. Kreuk et al. (2018) used a very similar methodology, collecting 7,150 benign executables from a fresh Windows 8.1 installation and using “ninite” to supplement that with executables from over 50 other software vendors. They used 10,866 malicious executables from the Microsoft Kaggle 2015 dataset. Due to the small dataset sizes and similar collection methodologies used in other failed experiments, it seems likely that datasets used by both Chen et al. (2019) and Kreuk et al. (2018) suffer from overfitting issues as well.

Due to the numerous issues with malware dataset collection in related research, this dissertation will attempt to draw samples from large datasets, where PE files are collected in the millions instead of thousands. Effort will be made to ensure that the samples are representative of the entire population of PE files and not biased towards a particular software vendor or malware family.

Due to compute restrictions and a short timeline for project completion, the EMBER-trained MalConv model will be used. The EMBER dataset contains more than one million Windows PE files (Anderson & Roth, 2018). The training phase is when it becomes most necessary to use large datasets to ensure the model can generalize well. Running samples through the resulting model for experimental purposes likely doesn’t require millions of samples to be used, but the samples should at least be representative of the entire population of Windows PE files. By drawing uniformly at random from a dataset with millions of Windows PE files, it’s unlikely for the experiment to be biased towards a particular type or family of malware.

One challenge for this project was the lack of publicly available datasets of the extreme sizes necessary to prevent bias and overfitting with this problem set. The EMBER dataset is frequently cited within the research literature, but samples shouldn’t be drawn from the same set of binaries that the model was trained with. Additionally, the EMBER dataset doesn’t make the actual binaries public, instead opting to release feature data (Anderson &
Roth, 2018). The dataset of 2 million binaries used by Raff et al. (2018a) does not appear to be publicly available, nor does the dataset of 12.5 million binaries used by Suciu et al. (2019). In both cases it appears likely that the research teams partnered with an antivirus vendor or security company.

The Sophos AI team later released a production-scale dataset called SOREL-20M, consisting of with over 20 million Windows PE files (Harang & Rudd, 2020). The SOREL-20M dataset contains over 10 million malicious executables made available via an AWS S3 bucket. Sophos disarmed all malware samples by altering the header flags such that the OptionalHeader.Subsystem flag and FileHeader.Machine flag are both set to 0. Each of the samples were then compressed using zlib before being uploaded to the S3 bucket.

Results reported in the following chapter are the result of experiments whose inputs were drawn uniformly at random from the complete pool of malicious Windows PE files from the SOREL-20M dataset. The samples all need to be uncompressed and rearmed to arrive at their original state. Leaving the binaries in a disarmed state did not appear to have a noticeable effect on results and were only re-armed for completeness and to ensure full accuracy of each experiment. Downloaded samples were immediately piped into the zlib-flate utility to decompress the samples.

Limitations and Assumptions

The authors of the original MalConv architecture proposed a model that allowed for executable files up to 2MB in length, using a batch size of 256 (Raff et al., 2018a). The authors of the EMBER dataset opted to use 1MB file sizes and batch sizes of 100 to fit within the resource limits of two Titan X (Pascal) GPUs (Anderson & Roth, 2018). To err on the side of caution, the attack artifact designed for this dissertation will assume similar constraints on memory and compute resources exist. An assumption is that the offensive and defensive techniques described in this dissertation would apply equally to the original MalConv model as well.

The primary limitation to the append-based FGSM attack is its inability to work on binaries greater than 1MB in size, the maximum input size allowed by the EMBER-trained MalConv model. Previous research has demonstrated that slack space attacks may be successful in filling this gap (Suciu et al., 2019).
**Expected Contributions**

Machine learning models have demonstrated incredible potential as compared to traditional antivirus products, especially in their ability to generalize and detect previously unseen variants of malware. One of their greatest flaws, however, is their susceptibility to attack via adversarial examples. Featureless raw-byte classifiers such as MalConv largely eliminate the need for domain expertise but are still vulnerable to adversarial attacks. Protecting machine learning models from adversarial examples is essential if classification models are to be used within production environments. A key motivation behind this offensive-oriented dissertation is to demonstrate the ease at which adversarial attacks can be employed, helping to eliminate the perception that the attacks are merely theoretical in nature.

Although detection models used within antivirus products would largely be considered black-box attack scenarios, the white-box attack algorithms presented within this dissertation are still quite practical. Yuan, et al. (2019) explain a common technique for attackers is to attack a white-box model trained with similar data and attempt to transfer the adversarial example to the black-box environment. Chakroborty et al. (2018) further explain that black-box environments can be repeatedly queried to gather input-output tuples, which can in-turn be used to train a surrogate model allowing for use of white-box attacks.

Specific contributions to the research literature include the release of an open-source attack tool, new payload initialization methods to select from, a payload size minimization algorithm, a near-instantaneous byte reconstruction phase, and the ability to bypass pre-detection mechanisms that may be included as part of a larger detection pipeline. The defeat of the pre-detection mechanism demonstrates how easily attacks can be chained together as a means of bypassing multiple detection capabilities. The significance of attacking this pre-detection mechanism is that many prior studies are focused exclusively on attacks against the model itself. Recent job postings suggest that companies are now hiring security architects and penetration testers to secure their AI/ML technologies, with scopes extending well beyond the machine learning model itself, to include MLOps pipelines and supporting infrastructure.
Summary

This chapter explained the research methodology that will be used for the remainder of this dissertation. A design science methodology was chosen with the goal of designing a viable solution to a relevant problem in the field. The contributions presented in this research will improve adversarial attacks against machine learning models and their associated pre-processing pipelines. An artifact was designed and fully implemented. The artifact will be used for experiments in the next chapter, where a large dataset will be used to compare treatment results. The expectation is that a statistically discernable difference exists when compared to previous attacks.
CHAPTER 4

RESULTS

After implementing the attack artifact described in chapter three, experiments were then conducted to determine its level of success in attacks against MalConv. The results assist in answering the original research questions outlined in chapter one. Results are presented in five distinct phases. The first phase runs the attack artifact using the new byte initialization techniques described in chapter three to determine which methods yield the highest evasion rates. The second phase analyzes speed improvements to the byte reconstruction phase after implementation of K-D tree queries and then verifies correctness of its results. The third phase attempts to defeat the pre-detection mechanism using a single byte attack to the code section. The fourth phase attempts to find the minimal payload size possible without making sacrifices to evasion rate, making use of the new payload size minimization algorithm proposed in the previous chapter. The final phase compares the new attack tool to other open-source libraries and results provided within the research literature.

Payload Initialization Strategies

The weighted payload initialization strategy is derived from the byte distribution counts of successful payloads, as described in the previous chapter. The byte distributions were gathered by selecting 1,000 samples at random from the SOREL-20M dataset. This dataset was not used for any other purpose, with validation of payload initialization strategies being performed using a different subset of samples. An attack was run against all 1,000 samples using the random payload initialization strategy. Perturbed samples that successfully evaded MalConv were then analyzed, appending the byte distribution counts of their payload to a global data structure containing cumulative counts. The most frequently occurring bytes were 0x00 (0), 0x3F (63), 0x68 (104), and 0x8B (139).
A new dataset of 500 new samples was produced by again selecting samples uniformly at random from the SOREL-20M dataset. The samples were then run through the attack tool for each of the proposed payload initialization techniques while using 1,000-byte payloads. The random initialization strategy is used by existing attacks within the research literature and serves as a baseline to compare new strategies against. With the constant byte value strategies, the entire payload is filled with the same byte. The separate experiment is run for all 256 possible byte values. Table 1 shows the resulting evasion percentages and the post-reconstruction MalConv scores.

Table 1. A comparison of payload initialization strategies.

<table>
<thead>
<tr>
<th>Initialization Strategy</th>
<th>MalConv Maliciousness Score (Post-Reconstruction)</th>
<th>Evasion Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg</td>
<td>Std</td>
</tr>
<tr>
<td>Random (Original)</td>
<td>53.27</td>
<td>48.51</td>
</tr>
<tr>
<td>Weighted</td>
<td>29.42</td>
<td>44.03</td>
</tr>
<tr>
<td>Byte 0x00 (0)</td>
<td>42.75</td>
<td>48.52</td>
</tr>
<tr>
<td>Byte 0x01 (1)</td>
<td>43.21</td>
<td>49.29</td>
</tr>
<tr>
<td>Byte 0x02 (2)</td>
<td>61.16</td>
<td>47.64</td>
</tr>
<tr>
<td>Byte 0x03 (3)</td>
<td>15.03</td>
<td>35.38</td>
</tr>
<tr>
<td>Byte 0x04 (4)</td>
<td>66.78</td>
<td>45.53</td>
</tr>
<tr>
<td>Byte 0x05 (5)</td>
<td>77.69</td>
<td>39.63</td>
</tr>
<tr>
<td>Byte 0x06 (6)</td>
<td>46.53</td>
<td>49.60</td>
</tr>
<tr>
<td>Byte 0x07 (7)</td>
<td>21.66</td>
<td>41.11</td>
</tr>
<tr>
<td>Byte 0x08 (8)</td>
<td>15.33</td>
<td>32.99</td>
</tr>
<tr>
<td>Byte 0x09 (9)</td>
<td>48.66</td>
<td>49.24</td>
</tr>
<tr>
<td>Byte 0x0A (10)</td>
<td>38.71</td>
<td>48.52</td>
</tr>
<tr>
<td>Byte 0x0B (11)</td>
<td>36.48</td>
<td>47.92</td>
</tr>
<tr>
<td>Byte 0x0C (12)</td>
<td>10.48</td>
<td>29.08</td>
</tr>
<tr>
<td>Byte 0x0D (13)</td>
<td>75.64</td>
<td>40.98</td>
</tr>
<tr>
<td>Byte 0x0E (14)</td>
<td>53.16</td>
<td>48.29</td>
</tr>
<tr>
<td>Byte 0x0F (15)</td>
<td>37.66</td>
<td>48.15</td>
</tr>
<tr>
<td>Byte 0x10 (16)</td>
<td>23.40</td>
<td>42.38</td>
</tr>
<tr>
<td>Byte 0x11 (17)</td>
<td>42.75</td>
<td>49.47</td>
</tr>
<tr>
<td>Byte 0x12 (18)</td>
<td>15.19</td>
<td>35.01</td>
</tr>
<tr>
<td>Byte 0x13 (19)</td>
<td>11.63</td>
<td>31.90</td>
</tr>
<tr>
<td>Byte 0x14 (20)</td>
<td>5.80</td>
<td>22.12</td>
</tr>
<tr>
<td>Byte 0x15 (21)</td>
<td>0.92</td>
<td>7.18</td>
</tr>
<tr>
<td>Byte 0x16 (22)</td>
<td>16.13</td>
<td>36.19</td>
</tr>
<tr>
<td>Byte 0x17 (23)</td>
<td>15.67</td>
<td>35.84</td>
</tr>
<tr>
<td>Byte 0x18 (24)</td>
<td>68.27</td>
<td>45.26</td>
</tr>
<tr>
<td>Byte 0x19 (25)</td>
<td>36.70</td>
<td>47.65</td>
</tr>
<tr>
<td>Byte 0x1A (26)</td>
<td>32.92</td>
<td>45.93</td>
</tr>
<tr>
<td>Byte 0x1B (27)</td>
<td>55.07</td>
<td>48.75</td>
</tr>
<tr>
<td>Byte 0x1C (28)</td>
<td>33.55</td>
<td>44.64</td>
</tr>
<tr>
<td>Byte 0x1D (29)</td>
<td>30.02</td>
<td>43.75</td>
</tr>
<tr>
<td>Byte 0x1E (30)</td>
<td>60.36</td>
<td>46.05</td>
</tr>
<tr>
<td>Byte 0x1F (31)</td>
<td>55.06</td>
<td>47.97</td>
</tr>
<tr>
<td>Byte 0x20 (32)</td>
<td>26.54</td>
<td>43.18</td>
</tr>
<tr>
<td>Byte 0x21 (33)</td>
<td>55.79</td>
<td>49.44</td>
</tr>
<tr>
<td>Byte 0x22 (34)</td>
<td>59.89</td>
<td>48.73</td>
</tr>
<tr>
<td>Byte 0x23 (35)</td>
<td>74.11</td>
<td>41.12</td>
</tr>
<tr>
<td>Byte 0x24 (36)</td>
<td>70.37</td>
<td>45.59</td>
</tr>
<tr>
<td>Byte 0x25 (37)</td>
<td>64.76</td>
<td>46.66</td>
</tr>
<tr>
<td>Byte 0x26 (38)</td>
<td>48.00</td>
<td>49.27</td>
</tr>
<tr>
<td>Byte 0x27 (39)</td>
<td>59.22</td>
<td>47.35</td>
</tr>
<tr>
<td>Byte 0x28 (40)</td>
<td>53.75</td>
<td>49.50</td>
</tr>
<tr>
<td>Byte 0x29 (41)</td>
<td>61.34</td>
<td>48.47</td>
</tr>
<tr>
<td>Byte 0x2A (42)</td>
<td>48.77</td>
<td>49.20</td>
</tr>
<tr>
<td>Byte 0x2B (43)</td>
<td>31.81</td>
<td>46.61</td>
</tr>
<tr>
<td>Byte 0x2C (44)</td>
<td>24.69</td>
<td>42.79</td>
</tr>
<tr>
<td>Byte 0x2D (45)</td>
<td>69.50</td>
<td>45.62</td>
</tr>
<tr>
<td>Byte</td>
<td>Value 1</td>
<td>Value 2</td>
</tr>
<tr>
<td>------</td>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>0x2E</td>
<td>40.30</td>
<td>48.15</td>
</tr>
<tr>
<td>0x2F</td>
<td>18.24</td>
<td>37.07</td>
</tr>
<tr>
<td>0x30</td>
<td>54.11</td>
<td>47.64</td>
</tr>
<tr>
<td>0x31</td>
<td>38.68</td>
<td>48.29</td>
</tr>
<tr>
<td>0x32</td>
<td>63.27</td>
<td>47.54</td>
</tr>
<tr>
<td>0x33</td>
<td>55.57</td>
<td>49.12</td>
</tr>
<tr>
<td>0x34</td>
<td>70.82</td>
<td>45.49</td>
</tr>
<tr>
<td>0x35</td>
<td>18.23</td>
<td>38.32</td>
</tr>
<tr>
<td>0x36</td>
<td>65.95</td>
<td>47.28</td>
</tr>
<tr>
<td>0x37</td>
<td>54.40</td>
<td>48.08</td>
</tr>
<tr>
<td>0x38</td>
<td>50.40</td>
<td>49.19</td>
</tr>
<tr>
<td>0x39</td>
<td>70.12</td>
<td>44.25</td>
</tr>
<tr>
<td>0x3A</td>
<td>68.00</td>
<td>46.53</td>
</tr>
<tr>
<td>0x3B</td>
<td>58.02</td>
<td>48.10</td>
</tr>
<tr>
<td>0x3C</td>
<td>51.09</td>
<td>48.46</td>
</tr>
<tr>
<td>0x3D</td>
<td>45.27</td>
<td>46.91</td>
</tr>
<tr>
<td>0x3E</td>
<td>64.63</td>
<td>46.27</td>
</tr>
<tr>
<td>0x3F</td>
<td>29.01</td>
<td>44.56</td>
</tr>
<tr>
<td>0x40</td>
<td>10.20</td>
<td>30.30</td>
</tr>
<tr>
<td>0x41</td>
<td>57.23</td>
<td>49.50</td>
</tr>
<tr>
<td>0x42</td>
<td>69.01</td>
<td>45.21</td>
</tr>
<tr>
<td>0x43</td>
<td>47.12</td>
<td>48.49</td>
</tr>
<tr>
<td>0x44</td>
<td>23.97</td>
<td>41.48</td>
</tr>
<tr>
<td>0x45</td>
<td>53.62</td>
<td>48.40</td>
</tr>
<tr>
<td>0x46</td>
<td>27.49</td>
<td>44.20</td>
</tr>
<tr>
<td>0x47</td>
<td>7.90</td>
<td>26.68</td>
</tr>
<tr>
<td>0x48</td>
<td>62.60</td>
<td>47.31</td>
</tr>
<tr>
<td>0x49</td>
<td>69.50</td>
<td>44.08</td>
</tr>
<tr>
<td>0x4A</td>
<td>45.45</td>
<td>49.59</td>
</tr>
<tr>
<td>0x4B</td>
<td>59.93</td>
<td>48.13</td>
</tr>
<tr>
<td>Byte 0x4C (76)</td>
<td>11.33</td>
<td>30.43</td>
</tr>
<tr>
<td>Byte 0x4D (77)</td>
<td>69.47</td>
<td>44.48</td>
</tr>
<tr>
<td>Byte 0x4E (78)</td>
<td>60.71</td>
<td>47.78</td>
</tr>
<tr>
<td>Byte 0x4F (79)</td>
<td>2.68</td>
<td>13.93</td>
</tr>
<tr>
<td>Byte 0x50 (80)</td>
<td>69.47</td>
<td>44.48</td>
</tr>
<tr>
<td>Byte 0x51 (81)</td>
<td>21.67</td>
<td>40.97</td>
</tr>
<tr>
<td>Byte 0x52 (82)</td>
<td>36.46</td>
<td>48.13</td>
</tr>
<tr>
<td>Byte 0x53 (83)</td>
<td>3.80</td>
<td>19.13</td>
</tr>
<tr>
<td>Byte 0x54 (84)</td>
<td>42.77</td>
<td>48.59</td>
</tr>
<tr>
<td>Byte 0x55 (85)</td>
<td>61.24</td>
<td>47.59</td>
</tr>
<tr>
<td>Byte 0x56 (86)</td>
<td>57.40</td>
<td>49.50</td>
</tr>
<tr>
<td>Byte 0x57 (87)</td>
<td>16.77</td>
<td>36.55</td>
</tr>
<tr>
<td>Byte 0x58 (88)</td>
<td>6.82</td>
<td>24.28</td>
</tr>
<tr>
<td>Byte 0x59 (89)</td>
<td>45.06</td>
<td>49.48</td>
</tr>
<tr>
<td>Byte 0x5A (90)</td>
<td>23.03</td>
<td>41.79</td>
</tr>
<tr>
<td>Byte 0x5B (91)</td>
<td>15.20</td>
<td>35.70</td>
</tr>
<tr>
<td>Byte 0x5C (92)</td>
<td>54.23</td>
<td>49.55</td>
</tr>
<tr>
<td>Byte 0x5D (93)</td>
<td>41.34</td>
<td>49.12</td>
</tr>
<tr>
<td>Byte 0x5E (94)</td>
<td>69.61</td>
<td>46.01</td>
</tr>
<tr>
<td>Byte 0x5F (95)</td>
<td>10.42</td>
<td>29.96</td>
</tr>
<tr>
<td>Byte 0x60 (96)</td>
<td>34.65</td>
<td>45.72</td>
</tr>
<tr>
<td>Byte 0x61 (97)</td>
<td>48.40</td>
<td>50.02</td>
</tr>
<tr>
<td>Byte 0x62 (98)</td>
<td>59.82</td>
<td>48.49</td>
</tr>
<tr>
<td>Byte 0x63 (99)</td>
<td>57.60</td>
<td>48.71</td>
</tr>
<tr>
<td>Byte 0x64 (100)</td>
<td>28.78</td>
<td>44.75</td>
</tr>
<tr>
<td>Byte 0x65 (101)</td>
<td>70.84</td>
<td>45.08</td>
</tr>
<tr>
<td>Byte 0x66 (102)</td>
<td>26.12</td>
<td>43.19</td>
</tr>
<tr>
<td>Byte 0x67 (103)</td>
<td>57.81</td>
<td>48.28</td>
</tr>
<tr>
<td>Byte 0x68 (104)</td>
<td>65.60</td>
<td>47.55</td>
</tr>
<tr>
<td>Byte 0x69 (105)</td>
<td>27.40</td>
<td>44.64</td>
</tr>
<tr>
<td>Byte</td>
<td>Value 1</td>
<td>Value 2</td>
</tr>
<tr>
<td>-------</td>
<td>---------</td>
<td>---------</td>
</tr>
<tr>
<td>0x6A</td>
<td>12.36</td>
<td>31.45</td>
</tr>
<tr>
<td>0x6B</td>
<td>15.90</td>
<td>35.12</td>
</tr>
<tr>
<td>0x6C</td>
<td>47.94</td>
<td>49.60</td>
</tr>
<tr>
<td>0x6D</td>
<td>14.45</td>
<td>34.72</td>
</tr>
<tr>
<td>0x6E</td>
<td>41.24</td>
<td>48.96</td>
</tr>
<tr>
<td>0x6F</td>
<td>39.60</td>
<td>48.96</td>
</tr>
<tr>
<td>0x70</td>
<td>23.60</td>
<td>40.03</td>
</tr>
<tr>
<td>0x71</td>
<td>48.45</td>
<td>48.79</td>
</tr>
<tr>
<td>0x72</td>
<td>64.27</td>
<td>46.13</td>
</tr>
<tr>
<td>0x73</td>
<td>57.94</td>
<td>49.18</td>
</tr>
<tr>
<td>0x74</td>
<td>54.83</td>
<td>49.79</td>
</tr>
<tr>
<td>0x75</td>
<td>45.67</td>
<td>49.50</td>
</tr>
<tr>
<td>0x76</td>
<td>5.99</td>
<td>23.08</td>
</tr>
<tr>
<td>0x77</td>
<td>32.22</td>
<td>46.76</td>
</tr>
<tr>
<td>0x78</td>
<td>61.60</td>
<td>48.68</td>
</tr>
<tr>
<td>0x79</td>
<td>38.84</td>
<td>47.89</td>
</tr>
<tr>
<td>0x7A</td>
<td>35.00</td>
<td>47.74</td>
</tr>
<tr>
<td>0x7B</td>
<td>36.40</td>
<td>47.97</td>
</tr>
<tr>
<td>0x7C</td>
<td>16.83</td>
<td>37.41</td>
</tr>
<tr>
<td>0x7D</td>
<td>13.12</td>
<td>32.99</td>
</tr>
<tr>
<td>0x7E</td>
<td>13.51</td>
<td>33.58</td>
</tr>
<tr>
<td>0x7F</td>
<td>36.75</td>
<td>47.54</td>
</tr>
<tr>
<td>0x80</td>
<td>9.22</td>
<td>28.79</td>
</tr>
<tr>
<td>0x81</td>
<td>66.53</td>
<td>46.67</td>
</tr>
<tr>
<td>0x82</td>
<td>21.27</td>
<td>39.81</td>
</tr>
<tr>
<td>0x83</td>
<td>13.95</td>
<td>34.47</td>
</tr>
<tr>
<td>0x84</td>
<td>2.17</td>
<td>14.32</td>
</tr>
<tr>
<td>0x85</td>
<td>23.40</td>
<td>42.38</td>
</tr>
<tr>
<td>0x86</td>
<td>33.29</td>
<td>46.56</td>
</tr>
<tr>
<td>0x87</td>
<td>37.99</td>
<td>47.71</td>
</tr>
<tr>
<td>Byte</td>
<td>Value</td>
<td>Error</td>
</tr>
<tr>
<td>------</td>
<td>-------</td>
<td>-------</td>
</tr>
<tr>
<td>0x88</td>
<td>35.09</td>
<td>46.67</td>
</tr>
<tr>
<td>0x89</td>
<td>29.88</td>
<td>45.59</td>
</tr>
<tr>
<td>0x8A</td>
<td>36.61</td>
<td>46.53</td>
</tr>
<tr>
<td>0x8B</td>
<td>40.75</td>
<td>47.43</td>
</tr>
<tr>
<td>0x8C</td>
<td>18.25</td>
<td>37.05</td>
</tr>
<tr>
<td>0x8D</td>
<td>44.02</td>
<td>48.81</td>
</tr>
<tr>
<td>0x8E</td>
<td>44.52</td>
<td>45.85</td>
</tr>
<tr>
<td>0x8F</td>
<td>26.16</td>
<td>42.45</td>
</tr>
<tr>
<td>0x90</td>
<td>68.01</td>
<td>45.67</td>
</tr>
<tr>
<td>0x91</td>
<td>56.97</td>
<td>49.23</td>
</tr>
<tr>
<td>0x92</td>
<td>77.54</td>
<td>39.62</td>
</tr>
<tr>
<td>0x93</td>
<td>64.06</td>
<td>46.74</td>
</tr>
<tr>
<td>0x94</td>
<td>57.10</td>
<td>47.59</td>
</tr>
<tr>
<td>0x95</td>
<td>52.54</td>
<td>48.04</td>
</tr>
<tr>
<td>0x96</td>
<td>67.89</td>
<td>44.64</td>
</tr>
<tr>
<td>0x97</td>
<td>72.10</td>
<td>43.01</td>
</tr>
<tr>
<td>0x98</td>
<td>67.04</td>
<td>45.68</td>
</tr>
<tr>
<td>0x99</td>
<td>56.60</td>
<td>45.72</td>
</tr>
<tr>
<td>0x9A</td>
<td>6.31</td>
<td>21.82</td>
</tr>
<tr>
<td>0x9B</td>
<td>35.64</td>
<td>47.58</td>
</tr>
<tr>
<td>0x9C</td>
<td>25.46</td>
<td>42.59</td>
</tr>
<tr>
<td>0x9D</td>
<td>46.00</td>
<td>48.85</td>
</tr>
<tr>
<td>0x9E</td>
<td>10.67</td>
<td>30.43</td>
</tr>
<tr>
<td>0x9F</td>
<td>69.01</td>
<td>44.38</td>
</tr>
<tr>
<td>0xA0</td>
<td>7.37</td>
<td>23.61</td>
</tr>
<tr>
<td>0xA1</td>
<td>23.48</td>
<td>41.10</td>
</tr>
<tr>
<td>0xA2</td>
<td>15.94</td>
<td>35.85</td>
</tr>
<tr>
<td>0xA3</td>
<td>36.17</td>
<td>46.37</td>
</tr>
<tr>
<td>0xA4</td>
<td>12.68</td>
<td>31.08</td>
</tr>
<tr>
<td>0xA5</td>
<td>30.66</td>
<td>45.42</td>
</tr>
<tr>
<td>Byte 0xA6 (166)</td>
<td>47.25</td>
<td>49.93</td>
</tr>
<tr>
<td>Byte 0xA7 (167)</td>
<td>51.82</td>
<td>47.72</td>
</tr>
<tr>
<td>Byte 0xA8 (168)</td>
<td>67.84</td>
<td>45.70</td>
</tr>
<tr>
<td>Byte 0xA9 (169)</td>
<td>56.43</td>
<td>49.20</td>
</tr>
<tr>
<td>Byte 0xAA (170)</td>
<td>1.79</td>
<td>12.25</td>
</tr>
<tr>
<td>Byte 0xAB (171)</td>
<td>3.65</td>
<td>18.28</td>
</tr>
<tr>
<td>Byte 0xAC (172)</td>
<td>14.48</td>
<td>34.88</td>
</tr>
<tr>
<td>Byte 0xAD (173)</td>
<td>22.84</td>
<td>41.50</td>
</tr>
<tr>
<td>Byte 0xAE (174)</td>
<td>67.84</td>
<td>45.70</td>
</tr>
<tr>
<td>Byte 0xAF (175)</td>
<td>56.43</td>
<td>49.20</td>
</tr>
<tr>
<td>Byte 0xB0 (176)</td>
<td>1.79</td>
<td>12.25</td>
</tr>
<tr>
<td>Byte 0xB1 (177)</td>
<td>3.65</td>
<td>18.28</td>
</tr>
<tr>
<td>Byte 0xB2 (178)</td>
<td>14.48</td>
<td>34.88</td>
</tr>
<tr>
<td>Byte 0xB3 (179)</td>
<td>22.84</td>
<td>41.50</td>
</tr>
<tr>
<td>Byte 0xB4 (180)</td>
<td>67.84</td>
<td>45.70</td>
</tr>
<tr>
<td>Byte 0xB5 (181)</td>
<td>56.43</td>
<td>49.20</td>
</tr>
<tr>
<td>Byte 0xB6 (182)</td>
<td>1.79</td>
<td>12.25</td>
</tr>
<tr>
<td>Byte 0xB7 (183)</td>
<td>3.65</td>
<td>18.28</td>
</tr>
<tr>
<td>Byte 0xB8 (184)</td>
<td>14.48</td>
<td>34.88</td>
</tr>
<tr>
<td>Byte 0xB9 (185)</td>
<td>22.84</td>
<td>41.50</td>
</tr>
<tr>
<td>Byte 0xBA (186)</td>
<td>67.84</td>
<td>45.70</td>
</tr>
<tr>
<td>Byte 0xBB (187)</td>
<td>56.43</td>
<td>49.20</td>
</tr>
<tr>
<td>Byte 0xBC (188)</td>
<td>1.79</td>
<td>12.25</td>
</tr>
<tr>
<td>Byte 0xBD (189)</td>
<td>3.65</td>
<td>18.28</td>
</tr>
<tr>
<td>Byte 0xBE (190)</td>
<td>14.48</td>
<td>34.88</td>
</tr>
<tr>
<td>Byte 0xBF (191)</td>
<td>22.84</td>
<td>41.50</td>
</tr>
<tr>
<td>Byte 0xC0 (192)</td>
<td>67.84</td>
<td>45.70</td>
</tr>
<tr>
<td>Byte 0xC1 (193)</td>
<td>56.43</td>
<td>49.20</td>
</tr>
<tr>
<td>Byte 0xC2 (194)</td>
<td>1.79</td>
<td>12.25</td>
</tr>
<tr>
<td>Byte 0xC3 (195)</td>
<td>3.65</td>
<td>18.28</td>
</tr>
<tr>
<td>Byte 0xC4 (196)</td>
<td>46.07</td>
<td>49.66</td>
</tr>
<tr>
<td>Byte 0xC5 (197)</td>
<td>27.34</td>
<td>43.39</td>
</tr>
<tr>
<td>Byte 0xC6 (198)</td>
<td>24.44</td>
<td>42.98</td>
</tr>
<tr>
<td>Byte 0xC7 (199)</td>
<td>60.24</td>
<td>48.71</td>
</tr>
<tr>
<td>Byte 0xC8 (200)</td>
<td>60.86</td>
<td>47.62</td>
</tr>
<tr>
<td>Byte 0xC9 (201)</td>
<td>46.64</td>
<td>49.55</td>
</tr>
<tr>
<td>Byte 0xCA (202)</td>
<td>49.78</td>
<td>48.55</td>
</tr>
<tr>
<td>Byte 0xCB (203)</td>
<td>38.59</td>
<td>47.16</td>
</tr>
<tr>
<td>Byte 0xCC (204)</td>
<td>55.05</td>
<td>47.98</td>
</tr>
<tr>
<td>Byte 0xCD (205)</td>
<td>71.97</td>
<td>43.22</td>
</tr>
<tr>
<td>Byte 0xCE (206)</td>
<td>40.10</td>
<td>48.23</td>
</tr>
<tr>
<td>Byte 0xCF (207)</td>
<td>9.77</td>
<td>29.66</td>
</tr>
<tr>
<td>Byte 0xD0 (208)</td>
<td>48.37</td>
<td>49.45</td>
</tr>
<tr>
<td>Byte 0xD1 (209)</td>
<td>63.63</td>
<td>46.64</td>
</tr>
<tr>
<td>Byte 0xD2 (210)</td>
<td>38.24</td>
<td>48.40</td>
</tr>
<tr>
<td>Byte 0xD3 (211)</td>
<td>17.42</td>
<td>37.46</td>
</tr>
<tr>
<td>Byte 0xD4 (212)</td>
<td>41.62</td>
<td>48.61</td>
</tr>
<tr>
<td>Byte 0xD5 (213)</td>
<td>31.89</td>
<td>45.78</td>
</tr>
<tr>
<td>Byte 0xD6 (214)</td>
<td>44.51</td>
<td>47.78</td>
</tr>
<tr>
<td>Byte 0xD7 (215)</td>
<td>8.57</td>
<td>26.87</td>
</tr>
<tr>
<td>Byte 0xD8 (216)</td>
<td>8.73</td>
<td>27.61</td>
</tr>
<tr>
<td>Byte 0xD9 (217)</td>
<td>6.15</td>
<td>22.34</td>
</tr>
<tr>
<td>Byte 0xDA (218)</td>
<td>46.97</td>
<td>48.25</td>
</tr>
<tr>
<td>Byte 0xDB (219)</td>
<td>45.02</td>
<td>49.71</td>
</tr>
<tr>
<td>Byte 0xDC (220)</td>
<td>30.39</td>
<td>44.96</td>
</tr>
<tr>
<td>Byte 0xDD (221)</td>
<td>20.23</td>
<td>38.55</td>
</tr>
<tr>
<td>Byte 0xDE (222)</td>
<td>22.43</td>
<td>40.81</td>
</tr>
<tr>
<td>Byte 0xDF (223)</td>
<td>9.93</td>
<td>29.74</td>
</tr>
<tr>
<td>Byte 0xE0 (224)</td>
<td>50.44</td>
<td>49.05</td>
</tr>
<tr>
<td>Byte 0xE1 (225)</td>
<td>2.29</td>
<td>13.75</td>
</tr>
<tr>
<td>Byte</td>
<td>0xE2 (226)</td>
<td>0xE3 (227)</td>
</tr>
<tr>
<td>--------</td>
<td>------------</td>
<td>------------</td>
</tr>
<tr>
<td></td>
<td>45.13</td>
<td>37.82</td>
</tr>
<tr>
<td></td>
<td>47.80</td>
<td>48.37</td>
</tr>
</tbody>
</table>
The randomized payload initialization strategy used by previous attacks in the research literature results in a low evasion rate of 46.40%. The weighted initialization strategy provides a significant improvement, producing a 24.8% increase in evasion rate. When filling the payload with a constant value for initialization, byte 0xBF (191) performs best, producing an astonishing evasion rate of 99.60%. MalConv was only able to detect 2 of the 500 samples after the adversarial payload was applied. Byte value 0x05 (5) performed worst, producing an evasion rate of only 22.00%. While certain byte values may naturally be more indicative of maliciousness than others, note that these byte values are only used to initialize the payload, and do not necessarily represent byte values that will be present in the final payload after the gradient attack and reconstruction phase have been completed.

Most perturbed malware samples result in maliciousness scores very close to 0 or 100, indicating that most attacks are either overwhelmingly successful or complete failures. This is the primary reason why many experiments showed median scores of either 0 or 100. When the attack fails to lower the maliciousness scores, FGSM is likely encountering gradient values close to 0, resulting in extremely small perturbations, if any at all.

Given the determination of best performing byte initializations was performed on a dataset of only 500 samples, a follow-on experiment was completed using a much larger dataset of 5,000 samples. The top 8 candidates from the previous experiment were compared to see if the results held for this much larger dataset, again using 1,000-byte payloads. The results are provided below in Table 2.

Table 2. A comparison of the top eight payload initialization strategies.

<table>
<thead>
<tr>
<th>Initialization Strategy</th>
<th>MalConv Maliciousness Score (Post-Reconstruction)</th>
<th>Evasion Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg</td>
<td>Std</td>
</tr>
<tr>
<td>Byte 0x15 (21)</td>
<td>1.02</td>
<td>9.98</td>
</tr>
<tr>
<td>Byte 0x4F (79)</td>
<td>2.74</td>
<td>14.26</td>
</tr>
<tr>
<td>Byte 0x53 (83)</td>
<td>3.60</td>
<td>18.62</td>
</tr>
<tr>
<td>Byte 0x84 (132)</td>
<td>2.54</td>
<td>15.45</td>
</tr>
<tr>
<td>Byte 0xAA (170)</td>
<td>1.66</td>
<td>11.77</td>
</tr>
<tr>
<td>Byte 0xBF (191)</td>
<td>0.66</td>
<td>6.01</td>
</tr>
<tr>
<td>Byte 0xE1 (225)</td>
<td>2.01</td>
<td>12.51</td>
</tr>
<tr>
<td>Byte 0xF0 (240)</td>
<td>3.73</td>
<td>17.14</td>
</tr>
</tbody>
</table>
Results of the follow-on experiment reinforce that byte 0xBF (191) is the best payload initialization strategy. With a few other bytes also scoring evasion rates above 98%, additional experiments using even larger datasets may yield slightly different results. For the remainder of the dissertation, byte 0xBF will be used as the default payload initialization strategy for BitCamo.

**Evading the Pre-Detection Mechanism**

Before and after hashes of both the code section and full file are performed to determine if the pre-detection mechanism could potentially succeed with the given sample. The assumption is then made that the defender has previously observed all samples within the test dataset, giving the pre-detection mechanism an implied initial success rate of 100%, consistent with the results reported by Chen et al. (2019). A real-world implementation would naturally see lower success rates as the result of previously unseen malware entering the detection pipeline. The goal is to improve from this position by producing successful evasions. A successful evasion can be assumed when the code section hash is different than that of the original sample.

The subset of 1000 samples from the SOREL-20M dataset are run through the attack tool. As discussed in the previous chapter, the attack tool no longer makes any assumption that the code section is named `.text`, opting to use header values to dynamically determine the code section for a given executable.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Evasion Count</th>
<th>Evasion Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>No evasive measures taken</td>
<td>0</td>
<td>0.00%</td>
</tr>
<tr>
<td>Single byte attack to the <code>.text</code> section</td>
<td>608</td>
<td>60.80%</td>
</tr>
<tr>
<td>Single byte attack to a dynamically determined code section</td>
<td>636</td>
<td>63.60%</td>
</tr>
</tbody>
</table>

Results demonstrate that the technique offers an effective attack for evading the pre-detection mechanism. Failure cases stem from files with no available slack space at the end of their code section. To defeat the pre-detection mechanism in those instances, it seems
unavoidable that future research should be designed to target content within the code section using techniques such replacement of instructions with semantically equivalent instructions of equivalent length.

**Performance Improvements to the Byte Reconstruction Phase**

The attack artifact features three notable improvements over the originally proposed attack. First, only bytes within the payload are mapped backwards through the embedding layer, preventing wasted iterations on parts of the file that were not modified during the gradient attack. In a worst-case scenario where 1 MB files are used, this improvement could result in one thousand bytes being processed instead of one million bytes. Second, CPU parallelization was implemented to take advantage of all available hardware and compute power. Third, the L₂ distance norm calculations were replaced by queries to a K-D tree to find the closest byte to each perturbed 8-D embedding.

The table below demonstrates how each improvement affects runtime of the reconstruction phase. Results were obtained by running experiments against the same executable on the same hardware. A 340 KB executable was selected for use as the baseline, as it provides a very average and representative file size. A 5,000-byte attack payload was selected, creating a final perturbed executable size of 353,160 bytes. The experiments were run using an Intel i7-8750H CPU, containing 6 cores at 2.20 GHz. Another executable slightly smaller than the maximum 1 MB size was also used, such that a 5,000-byte payload could still be appended. This data point was included as a means of estimating the theoretical maximum runtime that can be encountered, as runtime is linearly dependent on file size with no apparent variation possible due to other characteristics of the executable. This maximum runtime will only vary depending on the compute power of the machine being used.
Table 4. Performance comparison of reconstruction methods.

<table>
<thead>
<tr>
<th>Payload Only</th>
<th>Parallel</th>
<th>K-D Tree</th>
<th>Runtime (340 KB File)</th>
<th>Runtime (1 MB File)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>06h 47m 15s</td>
<td>20h 19m 14s</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td>01h 55m 58s</td>
<td>05h 32m 32s</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td></td>
<td>06m 21s</td>
<td>06m 32s</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td></td>
<td>02m 11s</td>
<td>05m 48s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>X</td>
<td>01m 18s</td>
<td>04m 11s</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td></td>
<td>01m 28s</td>
<td>01m 32s</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
<td>X</td>
<td>02s</td>
<td>02s</td>
</tr>
<tr>
<td>X</td>
<td></td>
<td>X</td>
<td>01s</td>
<td>01s</td>
</tr>
</tbody>
</table>

When utilizing the payload-only strategy, the runtime necessary to attack the smaller executable was roughly the same as the runtime needed to attack the larger file. This result was fully expected, as in both cases the file size is irrelevant, as the reconstruction phase iterates over the 5,000-byte payload only.

Using K-D tree queries in combination with the payload-only strategy cut the worst-case runtime from over 20 hours to a mere 1 second. Given the tremendous amount of time necessary to run the original reconstruction algorithm against a single sample, it was deemed impractical and unnecessary to conduct each these experiments multiple times over a larger sample size. Instead, only the optimal attack utilizing K-D trees over the payload was run against a larger dataset of 5,000 samples. A follow-on experiment was therefore performed as a means of demonstrating consistency and to ensure the 1 second reconstruction duration was not an anomaly. Results showed a 1 second mean, 1 second median, and 2 second maximum, confirming the previously presented results were not artifacts of the two samples selected for the original experiment. The ability to run the reconstruction phase in a single second will have a profound impact on practical attacks against MalConv and the ability for researchers to test new gradient-based attack strategies over larger samples sizes.

An important but unexpected finding is the optimal result having not used the parallel processing strategy. The best runtime occurred with the payload-only technique in combination with a K-D tree query, completing in one second. The addition of parallel
processing increased the duration to two seconds. In all cases the K-D tree strategy was slower when parallelism was added. This occurs due to the time necessary to spawn and join threads taking longer than the K-D query itself. The K-D tree already offers blazing fast spatial searches, making the use of parallelism a hinderance to overall performance. The use of parallelism is therefore only recommended when using the L2 distance metric. As a result of this finding the use of parallel processing is no longer enabled by default within the attack tool. Its use is instead made completely optional, enabled via use of a command line flag. Alternatively, parallelism could have instead been implemented such that each sample runs in its own thread as opposed to parallelizing the reconstruction phase only.

**Validation of K-D Tree Query Accuracy**

Given the massive speed improvements produced by using K-D tree queries over the L2 distance metric, the next natural step is to ensure the speed improvements didn’t come at the cost of reduced accuracy. Verification that both strategies produce the same or similar results is necessary before K-D trees can gain acceptance as a full replacement for the reconstruction phase of the attack.

Using the same hardware described in the previous section, experiments were performed on a much larger sample size of 1,000 executables, all chosen at random from the SOREL-20M dataset. Reconstruction was always performed using the payload-only strategy with parallelism enabled. Smaller payloads of 1,000 bytes were used with the random payload initialization strategy to prevent all samples from producing successful evasions and to allow for more variance in results. Validation was performed by modifying the attack algorithm to perform the following steps:

- Running the attack tool normally until the gradient attack phase was completed
- Running both reconstruction methods (K-D trees and L2 norms) against the perturbed embeddings
- Comparing the resulting payloads using a SHA-256 hash.

In this experiment, all 1,000 executables produced identical payloads after both reconstruction methods were run. While this experiment does not necessarily prove that both methods produce the same output for all possible inputs, it does provide an adequate level of
validation necessary to show that the K-D tree data structure is safe to use as a replacement for the reconstruction phase.

**Achieving High Evasion Rates and Minimizing Payload Sizes**

By utilizing all improvements presented within this dissertation, this phase of the research attempts to achieve the highest possible evasion rates while minimizing the payload sizes. The first experiment in this phase uses new dataset of 5,000 executables, again pulled uniformly at random from the SOREL-20M dataset. Processing larger dataset sizes is made possible by the performance improvements offered by K-D tree queries. The optimal byte value 0xBF (191) was be used for the payload initialization strategy, as discovered earlier in the chapter. FGSM was then run across the embedded payload, and the final payload was reconstructed using K-D tree queries. Several fixed-size payload lengths were attempted to determine the smallest possible payload necessary to attain a near-perfect evasion rate.

The results of this experiment can be found in Table 5. An evasion rate of 99% is attained using 900-byte payloads. The experiment increases the payload size in 100-byte increments, running the attack tool against the full dataset for each payload size. The attack continues to yield better evasion results until 900 bytes is reached, at which point the attack is unable to improve any further.

Table 5. Evaluation of attack success using 0xBF (191) byte initialization.

<table>
<thead>
<tr>
<th>Payload Length</th>
<th>MalConv Maliciousness Score (Post-Reconstruction)</th>
<th>Evasion Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Avg</td>
<td>Std</td>
</tr>
<tr>
<td>100 bytes</td>
<td>75.89</td>
<td>40.46</td>
</tr>
<tr>
<td>200 bytes</td>
<td>53.24</td>
<td>46.91</td>
</tr>
<tr>
<td>300 bytes</td>
<td>34.69</td>
<td>45.01</td>
</tr>
<tr>
<td>400 bytes</td>
<td>19.56</td>
<td>36.51</td>
</tr>
<tr>
<td>500 bytes</td>
<td>8.52</td>
<td>25.82</td>
</tr>
<tr>
<td>600 bytes</td>
<td>4.33</td>
<td>18.91</td>
</tr>
<tr>
<td>700 bytes</td>
<td>2.47</td>
<td>14.70</td>
</tr>
<tr>
<td>800 bytes</td>
<td>1.47</td>
<td>11.64</td>
</tr>
</tbody>
</table>
It can be observed that nearly half of the samples are able to achieve evasion using payload sizes of only 200 bytes. Applying a fixed-length payload to the entire dataset is therefore not an optimal approach. Kreuk et al. (2018) provide an equation for automatically selecting a payload size. Its use reduces the average payload size to 871 bytes on average, but also reduces the evasion rate by a few percentage points, making the fixed 900-byte payloads a better overall choice. Given the massive speed improvements made possible through the use of K-D trees in the reconstruction phase, it becomes apparent that the attack can be attempted using several candidate payload sizes for each sample. This would greatly reduce the average payload size for successful attacks.

A binary search is an obvious choice for finding the optimal payload size in logarithmic time. In practice, however, this failed to work properly. During implementation it was discovered that some payload sizes can fail with large payloads but later succeed with smaller payloads. This presents problems for the binary search, as increase the payload size upon failure could be a move in the wrong direction for a certain subset of samples. Instead, the search for an optimal payload size was implemented as an iterative approach starting at 50-byte payloads and increasing in 50-byte increments up to a 2000-byte maximum payload size. These amounts can be adjusted depending on tolerance for longer runtimes.

The new payload size optimization algorithm was included in the attack tool and ran against the same 5,000 sample dataset as before. Results showed a 100% evasion rate with an average payload size of exactly 300 bytes. The reduction in payload size was expected, but the increase in evasion rate was not. Attempting several payload sizes allowed the remaining
stubborn samples to achieve evasion. As described earlier, some samples were able to benefit from a reduction in payload size rather than an increase. The new attack was then performed on a second dataset of 5,000 samples, again pulled uniformly at random from the larger SOREL-20M dataset. Results again showed a 100% evasion rate with just shy of 300-byte payload sizes on average. It’s possible that adding a binary search only after successful completion of this iterative process could allow for another very minor reduction in average payload size, effectively searching the 50-byte window between the last failed attempt and the most recent successful attempt. This will likely result in the average payload size dropping below 300 bytes.

**Comparison to Existing Attacks**

Two other open-source libraries for creating adversarial attacks against MalConv are SecML Malware by Demetrio et al. (2021b) and MalwareRL by Anderson et al. (2018). SecML Malware includes several white-box attacks that share similarities to the attacks presented in this dissertation. By contrast, MalwareRL is a black-box attack tool utilizing reinforcement learning. This section will focus exclusively on comparisons to other white-box attacks.

The work presented in this dissertation is largely an adaptation and improvement of the append-based FGSM attack originally proposed by Kreuk et al. (2018), which was later implemented by Demetrio et al. (2021b) as part of the SecML Malware library. Comparison tests against attacks provided in the SecML Malware library are therefore provided as a means of confirming that the evasion rates and runtimes produced by BitCamo are an improvement over existing attacks against MalConv.

For these experiments, 200 samples were selected uniformly at random from the SOREL-20M dataset, with file sizes still capped at 1 MB. The dataset size was significantly reduced from previous experiments to allow for direct comparison of success levels to slower attacks. While performing these experiments, it became clear that some samples were achieving evasion only because MalConv was classifying them incorrectly to begin with, as MalConv assigned them a benign label before ever being attacked. After dropping these samples from inclusion in the results, it became clear that certain attacks were completely unsuccessful, unable to influence the predictions of MalConv. The results provided in Table 6
therefore do not include samples that MalConv classified as benign prior to the attack code running. Each attack was attempted using 1,000-, 5,000-, and 10,000-byte payloads. All attacks were run using SecML Malware version 0.2.4 with default parameters used unless otherwise specified. Tuning additional parameters may have resulted in changes to the runtimes and evasion rates if attempted.

Given that each tool can output a perturbed executable, results were cross validated using the opposing tool. For example, the executables output from BitCamo were supplied to SecML Malware for the purposes of recording the initial prediction only, ensuring both tools were in agreeance that a particular executable had successfully achieved evasion against MalConv.

Table 6. Comparison of BitCamo v1.1.0 to SecML Malware v0.2.4.

<table>
<thead>
<tr>
<th>Attack</th>
<th>Total Runtime (Avg.)</th>
<th>Evasion Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>SecML CKreukEvasion 1,000 bytes</td>
<td>23s</td>
<td>0.00</td>
</tr>
<tr>
<td>SecML CKreukEvasion 5,000 bytes</td>
<td>33s</td>
<td>0.00</td>
</tr>
<tr>
<td>SecML CKreukEvasion 10,000 bytes</td>
<td>01m 02s</td>
<td>0.00</td>
</tr>
<tr>
<td>SecML CSuciuEvasion 1,000 bytes</td>
<td>14s</td>
<td>0.00</td>
</tr>
<tr>
<td>SecML CSuciuEvasion 5,000 bytes</td>
<td>26s</td>
<td>0.00</td>
</tr>
<tr>
<td>SecML CSuciuEvasion 10,000 bytes</td>
<td>48s</td>
<td>0.00</td>
</tr>
<tr>
<td>SecML CPaddingEvasion 1,000 bytes</td>
<td>01m 32s</td>
<td>16.67</td>
</tr>
<tr>
<td>SecML CPaddingEvasion 5,000 bytes</td>
<td>01m 10s</td>
<td>16.67</td>
</tr>
<tr>
<td>SecML CPaddingEvasion 10,000 bytes</td>
<td>01m 10s</td>
<td>16.67</td>
</tr>
<tr>
<td>Malware</td>
<td>Optimized</td>
<td>Time</td>
</tr>
<tr>
<td>---------------------------------</td>
<td>-----------</td>
<td>-------</td>
</tr>
<tr>
<td>SecML CHeaderFieldsEvasion</td>
<td>False</td>
<td>14s</td>
</tr>
<tr>
<td>optimize_all_dos=True</td>
<td>True</td>
<td>15s</td>
</tr>
<tr>
<td>SecML CContentShiftingEvasion</td>
<td></td>
<td>21s</td>
</tr>
<tr>
<td>SecML CHeaderEvasion</td>
<td>False</td>
<td>17s</td>
</tr>
<tr>
<td>optimize_all_dos=True</td>
<td>True</td>
<td>13s</td>
</tr>
<tr>
<td>SecML CFormatExploitEvasion</td>
<td></td>
<td>03s</td>
</tr>
<tr>
<td>SecML CExtendDOSEvasion</td>
<td></td>
<td>02s</td>
</tr>
<tr>
<td>BitCamo</td>
<td></td>
<td>01s</td>
</tr>
<tr>
<td>1,000 bytes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BitCamo</td>
<td></td>
<td>01s</td>
</tr>
<tr>
<td>5,000 bytes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BitCamo</td>
<td></td>
<td>03s</td>
</tr>
<tr>
<td>10,000 bytes</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BitCamo</td>
<td></td>
<td>05s</td>
</tr>
<tr>
<td>Minimal Payload Size Search</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Kreuk attack included in the SecML Malware library is closest in similarity to the attack strategy used in BitCamo, as both techniques perform a gradient-based attack against a payload appended to the end of the file. Results show that the SecML Malware implementation of the Kreuk attack does not work at all. It’s unclear if this is due to an implementation issue, software bug, or limitations in the attack itself.

SecML Malware also includes several attacks that are fundamentally different, opting to attack headers near the beginning of Windows PE files. Many of the novel contributions prosed by the authors of SecML Malware were focused on header field manipulations (Demetrio et al., 2021b). These attacks proved to be incredibly effective, providing an evasion rate of 96.38%. Although BitCamo was able to achieve a 100% evasion rate, it’s likely that SecML Malware would outperform BitCamo if the dataset included samples greater than 1
MB, which BitCamo is unable to attack in its current form. BitCamo can likely be extended to attack other areas of the Windows PE file with very little effort, which would give it the ability to successfully attack larger file sizes. The direct manipulation of headers by SecML Malware may be easier for analysts with domain expertise to identify, especially if the MS-DOS stub is overwritten. This may be an important consideration if attacks are extended to include larger detection pipelines or actual antivirus products. Future research and enhancements to BitCamo will be discussed in Chapter 5.

Results demonstrate that BitCamo is a competitive tool capable of achieving extremely high evasion rates and at significantly faster speeds that other attacks. SecML Malware is a comparatively more robust framework offering a wider variety of attacks for users to select from, with some capable of achieving similar results as BitCamo. SecML Malware is currently the best tool for attacking files greater than 1MB in size. Both tools would likely benefit from the addition of techniques used by the other tool.

In addition to direct comparison tests against SecML Malware, BitCamo is also compared to the reported results from other papers within the research literature. Other papers featuring gradient-based white-box attacks against MalConv reported rather inconsistent results overall. These papers include:

- Kolosnjaji et al. (2018) achieve an evasion rate of 60% when using 10,000-byte payloads in the file overlay.
- Kreuk et al. (2018) report a 99% evasion rate with payload sizes between 500 and 1,000 bytes when using iterative FGSM on a payload in the file overlay. While the results are impressive, the dataset used to train MalConv and evaluate the attack may be unsuited for the task at hand. The Microsoft Kaggle 2015 dataset was used, containing samples from only 9 different malware families and with their PE headers completely removed (Ronen, Radu, Feuerstein, Yom-Tov, & Ahmadi, 2018). Kreuk et al. (2018) use approximately 7,000 samples from this dataset, re-adding headers using different malicious executables downloaded from VirusShare. Both Suciu et al. (2019) and Raff et al. (2018a) allude to extreme overfitting issues when training MalConv on datasets of this size, suggesting instead that datasets with hundreds of thousands of samples be used, if not millions, to best represent executables that may be encountered a real-world
situation. The SecML Malware implementation of this same attack performs extremely poorly when used on the much more robust EMBER-trained MalConv model, as demonstrated earlier in this section.

- The one-shot FGSM attack by Suciu et al. (2019) attains a 1% evasion rate on a small dataset, 33% evasion rate against the EMBER-trained MalConv model, and a 71% evasion rate against a model trained on a much larger dataset, all while using a 10,000-byte payload.

- Chen et al. (2019) reported extremely low success rates of 1-2% when using FGSM on payload sizes of up to 20,000 bytes. Their newly proposed Enhanced-BFA attack algorithm is reported to achieve 99% evasion using 20,000 bytes, but only 74% using 500 bytes. Chen et al. (2019) also use a smaller dataset with only 5,200 malicious samples to train their model and evaluate the attacks, and may suffer from the same overfitting issues as Kreuk et al. (2018).

BitCamo appears to significantly outperform each of the above attacks in terms of both evasion rate and minimizing the required payload size, achieving a 100% evasion rate for files less than 1 MB, using payload sizes of only 300 bytes on average. Attack duration and performance were typically not provided by the authors of other papers.
CHAPTER 5

CONCLUSION

This chapter highlights notable contributions presented within this dissertation, to include discussion about the applicability of these contributions to attacks against other models. Known limitations of the new contributions are provided, as well as future research directions and recommendations for improvement of the attack tool.

Contributions

An open-source attack tool was developed with the goal of aiding researchers in producing adversarial attacks against malware classifiers. The intention is that the availability of such tooling will raise awareness about adversarial attacks and demonstrate the ease at which these attacks can be conducted. The ability to detect malware using featureless machine learning models is a powerful capability, but care should be taken to protect these models from this new form of attack. The new attack tool created for this dissertation was named BitCamo and can be found at https://www.github.com/juburr/bitcamo.

The payload reconstruction phase results in a natural loss in precision, making it possible for the embedded representation of bytes to successfully evade MalConv, but then fail again after reconstruction is completed. A modified algorithm was introduced that runs FGSM with a higher epsilon value and then continues running it until the prediction reported by MalConv is less than 1%. A much stronger perturbation is generated by continuing the attack instead of stopping after a score of less than 50% is obtained, as proposed in the initial algorithm by Kreuk et al. (2018). Stronger perturbations of the embedded bytes made it more likely that the payload would continue to evade MalConv after the reconstruction phase was completed, resulting in higher evasion rates than previous works.

New payload initialization strategies were presented. The most promising strategies involved filling the payload with the same byte value in all positions. This dissertation finds that the byte value 0xBF (191) allows for the highest evasion rate, followed closely by bytes 0x15 (21), 0xE1 (225), and 0xAA (170). Previous researchers use random payload
initializations, which resulted in an evasion rate of only 46.40% using 1,000-byte payloads with the new attack tool. The new payload initialization strategy significantly increased the evasion rate to 99% using fixed payload sizes of 900 bytes when applied to executables less than 1MB in size. An algorithm for optimizing the payload size was then introduced, also helping to evade the remaining 1% of samples. With both improvements present, the new attack tool achieves a 100% evasion rate using payload sizes of only 300 bytes on average.

A faster method for conducting the reconstruction phase of the attack was presented. K-D trees appear to offer an overwhelming improvement over repeatedly calculating the L_2 norm. The K-D tree queries allow the reconstruction phase to finish in under one second, allowing for a more practical bulk processing capability, ultimately enabling experimentation that led to discovery of the other improvements. This demonstrates the importance of the engineering cycle within design science research.

Finally, a single-byte attack against the code section of executables was presented as a means of defeating a recently proposed pre-detection mechanism. The attack succeeds in 63.60% of cases. The attack can be chained together with the improved adversarial attack strategy to defeat a detection pipeline containing both the pre-detection mechanism and MalConv.

**Research Questions**

The original research questions will now be re-visited, answered, and compared the original hypotheses offered in chapter three.

**Research Question 1**
Can the append-based FGSM attack be re-implemented and improved to achieve consistently high evasion rates?

**Answer**
The FGSM overlay attack was successfully re-implemented and improved to achieve 100% evasion rates using an average payload size of only 300 bytes when applied to Windows PE files less than 1 MB in size. This result exceeded expectations, with the original hypothesis predicting a 99% evasion rate when using payloads up to several thousand bytes in length.
Research Question 2
Can the append-based FGSM attack benefit from a new byte initialization strategy?

Answer
The improved payload initialization strategy was the most important contribution used to achieve the perfect evasion rate. The weighted initialization scheme was predicted to provide the best results but instead offered only a modest improvement. Initializing every byte with the same value was predicted to offer poor results, but the results varied greatly depending on which byte value was used. Certain byte values provided extremely poor results, whereas others allowed for the perfect evasion rates when used with the new payload size minimization algorithm.

Research Question 3
Can the recently proposed pre-detection mechanism be completely bypassed?

Answer
The lack of slack space in the code section of certain executables was problematic for the proposed attack. The original hypothesis predicted nearly every sample could achieve evasion, but only 63% of samples were able to successfully defeat the pre-detection mechanism.

Research Question 4
Can performance of the long-running reconstruction phase of the append-based FGSM attack be improved?

Answer
Results were much better than anticipated. K-D tree queries offered massive speed improvements, allowing the reconstruction phase to complete in under a second on a personal laptop.
Applicability of Contributions to Other Models

As one of the first successful implementations of a raw byte classifier, MalConv is an obvious target for researchers studying adversarial attacks. It’s important to note that many components from the MalConv architecture are likely to be present within other raw byte models as well, especially as a means of contending with memory limitations on GPUs. For example, the use of 1D convolutional layers is important for reducing the large dimensionality of raw byte inputs and dealing with spatial invariance, given that a malicious feature may not necessarily occur at the exact same position within every file. Likewise, max pooling layers are likely to be used over average pooling because a malicious feature may only appear once within a given file, and an average pooling layer may inadvertently drown out a single strong activation.

Recall from the literature review that another successful raw byte model, AvastConv, was proposed by Krcál et al. (2018). Although this model was developed independently of MalConv, it still ended up sharing many of the same architectural components, such as the inclusion of a non-differentiable 8D embedding layer, 1D convolutional layers, a max pooling layer, and fully connected layers (Krcál et al., 2018). This was not by mistake, as these components each provide extremely desirable outcomes, making it likely that future raw byte classifiers will share similar designs.

To that end, the K-D tree improvement can be used on any model containing a non-differentiable embedding layer and is not limited to the malware domain. Depending on the application, however, care should be taken to avoid the “curse of dimensionality” discussed earlier in this dissertation. For raw byte classifiers with only 256 possible byte values to map back to, this should not be an issue. Likewise, any attack attempting to map backwards through such an embedding layer would be wise to follow the guidance provided earlier in this dissertation, by producing extremely strong perturbations within embedding space to allow for some natural loss in precision to occur after the reconstruction phase of the attack is completed.

The use of the 0xBF byte initialization strategy is unlikely to work for malware classifiers trained on a different architecture or with a vastly different dataset. The strategy used to find this magic byte, however, is very likely to work with other architectures as well. It can be formalized as an algorithm that progressively eliminates underperforming bytes from
contention as viable candidates. This would save compute cycles by preventing samples later in the dataset from being attacked using candidate bytes that have low success rates thus far, identifying the top performers much quicker than the method used in chapter 4 of this dissertation.

The iterative search for an optimal payload size can be applied to any other payload-based attack against malware classifiers. The use of payloads is more likely to occur with attacks targeting raw byte classifiers than those targeting feature-based classifiers though. The search for an optimal payload size naturally incurs additional runtime as well, meaning it may not be best suited for use with attacks that already have significantly high runtimes.

The single-byte attack against the pre-detection mechanism is a tailored attack unlikely to be reused for anything other than changing the hash of the code section. The idea of targeting multiple defensive resources within a single detection pipeline is worthy of additional study and inclusion within other attack tools.

**Applicability of Modern Adversarial Attacks to MalConv**

This dissertation makes use of FGSM for performing the gradient-based portion of the attack against MalConv. FGSM is one of the oldest and simplest gradient attacks in existence. More accurately, an iterative version of FGSM alternatively known as the BIM is used. This begs the question of whether a more powerful and modern attack could have been used instead.

FGSM was manually replaced with other attack algorithms provided by the CleverHans library. At the very least, this successfully demonstrates the ability to use newer attacks against MalConv. Recall that BitCamo is already able to achieve a 100% evasion rate using FGSM when targeting executables susceptible to an overlay-based attack. Swapping in newer attacks appeared to offer no improvement to overall success rates or a reduction in payload sizes, and only contributed towards significantly increasing the runtime of the attack. By contract, FGSM is extremely fast, with each iteration completing nearly instantly. Given the already perfect evasion rate, it appears undesirable to replace FGSM with a different attack algorithm unless the algorithm can either further reduce the payload size or contend with the non-differentiable embedding layer in a different manner. As it stands, other algorithms simply increase the runtime with no clear benefit. This bears similarities to model
selection for machine learning, where the latest and greatest deep neural network may not necessarily be the best tool for a given job.

BitCamo can likely be adapted into a more generic framework, architected to include two distinct software components. One component would handle specifics of the Windows PE file format, carving out space to use for payloads, forwarding bytes through the initial embedding layer, and conducting the final reconstruction phase. The second component of the software would be responsible for performing the actual gradient-based attack given a range of bytes that can safely be perturbed, with the attack algorithm being completely configurable. This would allow FGSM to be swapped out for other attack algorithms, hiding intricacies of the Windows PE file format for researchers more interested in the machine learning portion of the attack. Another advantage to handling file format issues separately is that the first software component could itself become a pluggable interface, replaceable with code to handle other executable file formats such as ELF, or even file formats belonging to completely different domains outside of the malware arena. These formats could then be attacked without having to modify the gradient-based portion of the attack framework. Domain experts would be able to make additions to this part of the framework without the need to know anything about machine learning. It should be noted that refactoring BitCamo into a generic framework has yet to be performed. Applying attack algorithms other than FGSM is worth some additional investigation in future research, however, especially after a more pluggable framework is developed.

Limitations and Future Research

The append-based attack to the overlay section will fail when file sizes exceed the maximum input size for MalConv. In the case of the EMBER-trained MalConv model, the limit is 1 MB. For adversarial attacks to succeed against larger files, perturbations must be targeted towards early parts of the file to ensure adversarial payloads will be retained after the file is truncated. While most malware samples are relatively small, the next logical step is to add additional capabilities to the attack tool, including the slack space attack by Suciu et al (2019), header-based attacks by Demetrio et al. (2021b), and code section modifications by Sharif, Lucas, Bauer, Reiter, & Shintre (2019). Each of these attacks can place adversarial bytes at much earlier locations within Windows PE files and thereby allow attacks to succeed
when using file sizes larger than 1 MB. Attacks within slack space could also be more successful than those utilizing the file overlay, as first demonstrated by Suciu et al. (2019). This may be due to headers from the following section being included in the same 500-byte convolutional window as the adversarial payload.

BitCamo does not have the ability to defeat signed executables, such as those using Microsoft Authenticode. This limitation was not a big impediment to the project, however, as the vast majority of malware is not signed.

The pre-detection mechanism is just one example of a tool or service that may be used to augment the MalConv detection model. The attack on the pre-detection mechanism was primarily included in this dissertation as a means of demonstrating that attacks can be chained together to defeat multiple detection capabilities with a single perturbed executable. Future research should consider even larger detection pipelines. Recent job postings reveal companies have an increased interest in applying red teaming resources towards AI models, MLOps pipelines, and even their supporting infrastructure. Likewise, academic research should also consider attacks that extend beyond the model itself.

Naturally, an end goal for an attack tool such as BitCamo is to demonstrate the ability to bypass real-world anti-virus products. Given that these products typically employ hybrid-based approaches driven by multiple detection capabilities, a single attack should therefore be capable of evading a diverse range of defenses. To that end, a natural next step for this project is to attack additional detection models instead of limiting the scope to MalConv. This should include additional raw byte classifiers such as non-negative MalConv, MalConv2, or AvastConv. More importantly, the tool must be capable of defeating feature-based detection models for which payload-based approaches will likely be less effective. A LightGBM model using EMBER features appears to be a popular approach for feature-based detection. Replacing TensorFlow with PyTorch may be beneficial in accomplishing some of these objectives, as the academic community now appears to heavily favor the use of PyTorch.

Other potential research directions may include:

- The ability to create adversarial examples using additional executable file formats, such as ELF and Mach-O, or even file formats frequently used to deliver malware, such as Microsoft Office documents. When attacking file formats without existing research literature pertaining to adversarial examples, a review of steganography,
data hiding, or covert channels for the associated file format would be highly beneficial, as similar methods will likely be used for placement of adversarial bytes within the file.

- Exploring optimal conditions that allow adversarial payloads to transfer to other executables, or research into the development of payloads that act as universal adversarial perturbations.
- Fine-grained code section modifications building off the work of Sharif et al. (2019) was an effective alternative to the much larger code caves being utilized at more obvious locations throughout the executable. This would naturally result in complete evasion of the pre-detection mechanism as well.
- Adversarial attacks against models used to classify the specific family of malware, such as models trained using the MOTIF dataset (Joyce, Amlani, Nicholas, & Raff, 2021). Misclassification of the malware family could have implications that include deliberate misattribution of malware to another threat actor.
REFERENCES


Krcál, M., Švec, O., Jasek, O., & Bálek, M. (2018). Deep convolutional malware classifiers can learn from raw executables and labels only.


