To the Graduate Council:

I am submitting herewith a dissertation written by Luke Ryan Koch entitled "EMBERs in the Dark: Countering AI-Based Malware Detection via Static Binary Instrumentation." I have examined the final paper copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Data Science and Engineering.

_____

Dr. Jeffrey A. Nichols, Major Professor

We have read this dissertation
and recommend its acceptance:

_____

Dr. Edmon Begoli

_____

Dr. Amir Sadovnik

_____

Dr. Sean Oesch

_____

Dr. Catherine Schuman

_____

Dr. Michael Jantz

Accepted for the Council:

_____

Dixie L. Thompson

Vice Provost and Dean of the Graduate School

To the Graduate Council:
I am submitting herewith a dissertation written by Luke Ryan Koch entitled "EMBERs in the Dark: Countering AI-Based Malware Detection via Static Binary Instrumentation." I have examined the final electronic copy of this dissertation for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy, with a major in Data Science and Engineering.

Dr. Jeffrey A. Nichols, Major Professor

We have read this dissertation
and recommend its acceptance:

Dr. Edmon Begoli

Dr. Amir Sadovnik

Dr. Sean Oesch

Dr. Catherine Schuman

Dr. Michael Jantz

Accepted for the Council:

Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

EMBERs in the Dark: Countering AI-Based Malware Detection via Static Binary Instrumentation

A Dissertation Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Luke Ryan Koch

August 2024

© by Luke Ryan Koch, 2024
All Rights Reserved.

For my folks who believed in me even when I shoveled rocks for a living.

Acknowledgements

I would like to thank Jeff, Amir, and Sean for meeting with me weekly. Amir taught me how to proceed in a principled manner rather than experimenting blindly. Sean taught me how to write for good conferences. Jeff kept me sane, which I rather selfishly consider supremely valuable. I would not have made it through the long slow recovery from the appendix rupture that damaged my lungs and liver without him. Edmon provided an example of leadership through inspiration rather than fear that I wish to emulate in my own career. I would also like to thank Cory Watson for setting up and maintaining the CAPE instances that played a critical role in my functionality verification of altered binaries. Jared Smith started me down the path to a dissertation and I'd like to thank him for his confidence in my abilities.

Machines don't fight wars. Terrain doesn't fight wars. People fight wars. It's in the minds of humans that wars must be fought.

Col. John Boyd

Abstract

Machine learning allows for the detection of novel malware. However, this method of detection introduces new vulnerabilities in the form of feature extraction evasion and adversarial instrumentation. These emerging methods for evading detection are hampered by the need to maintain functionality in altered binary files, a challenge largely unique to this domain. Functionality preservation is necessary to maintain the true label of altered files. Binary files, especially Windows Portable Executable files with an X86/86-64 architecture, may contain bytes whose role in functionality can only be determined via manual reverse-engineering. Therefore, automatic methods for altering these files without loss of function are highly restricted; we advance the state of research via the study of malware obfuscations and the development of novel actions. Since feature-extraction is a critical first step in malware classification, our first effort explored feature-extraction evasion and produced detection models to prevent this attacj. Having improved the reliability of feature-extraction evasion, we then demonstrated that input-space perturbations can still yield features that evade detection by commercial off-the-shelf tools without guidance. Given the large search space of possible input-space perturbations, a survey of guidance mechanisms was now warranted. This wide-spectrum survey revealed two key areas for improvement of adversarial machine learning agents that use static binary instrumentation to conduct functionality-preserving input-space perturbations. Finally, we demonstrated a set of novel actions paired with AI guidance and verified the effect these actions have on the evasion and functionality-preservation rate of altered binaries using strict criteria.

Table of Contents

1 Introduction **1**

1.1 Contribution: Chapter 2 4

1.2 Contribution: Chapter 3 6

1.3 Contribution: Chapter 4 7

1.4 Contribution: Chapter 5 8

2 Polyglot Creation and Detection **11**

2.1 Contribution Summary 11

2.2 Toward the Detection of Polyglot Files 12

2.2.1 Introduction 12

2.2.1.1 Overview 12

2.2.1.2 Motivation 13

2.2.2 Background 15

2.2.2.1 Polyglot Mechanics 15

2.2.2.1.1 Stacks 15

2.2.2.1.2 Parasites 16

2.2.2.1.3 Zippers 16

2.2.2.1.4 Cavities 17

2.2.2.2 Related Work 17

2.2.2.2.1 Related Tools 17

2.2.2.2.2 Related Research 18

2.2.2.3 Relation to This Work . . . 21
2.2.3 Methodology . . . 21
2.2.3.1 Polyglot Data Set . . . 21
2.2.3.2 Model Selection . . . 22
2.2.3.3 Train/Test Split and Hyperparameter Tuning . . . 24
2.2.4 Results & Discussion . . . 24
2.2.4.1 Testing Existing Tools . . . 24
2.2.4.2 Testing Machine Learning and Deep Learning . . . 25
2.2.4.3 Comparing ML and DL . . . 34
2.2.4.4 Detection on Novel Polyglots . . . 34
2.2.5 Conclusion and Future Work . . . 35
2.3 On the Abuse and Detection of Polyglot Files . . . 36
2.3.1 Introduction . . . 36
2.3.2 Related Work . . . 41
2.3.2.1 Polyglot Detection . . . 41
2.3.2.2 Polyglot Creation . . . 41
2.3.2.3 Polyglot Exploitation . . . 42
2.3.3 **RQ1:** Polyglot Exploitation in the Wild . . . 43
2.3.3.1 Survey Methods . . . 43
2.3.3.2 Role of Polyglot Files in Cyber Attack Chains . . . 44
2.3.3.2.1 IcedID . . . 45
2.3.3.2.2 Andariel/Lazarus . . . 46
2.3.4 Wild Polyglots: A Polyglot Data Set Based on Malicious Usage in the Wild . . . 48
2.3.4.1 Fazah: A Polyglot Generation Framework . . . 48
2.3.4.1.1 Wild Polyglots Data Set Creation and Contents 49
2.3.5 **RQ2:** Using Machine Learning for Polyglot Detection . . . 49
2.3.5.1 Ml-based Detection Development . . . 49
2.3.5.2 Comparing ML-based Polyglot Detection Approaches 53

2.3.6 **RQ3:** Comparison to Existing Signature-based File-format Identification Tools . . . 55
2.3.6.1 Tools Tested . . . 56
2.3.6.1.1 Binary Comparison . . . 56
2.3.6.1.2 Multi-label Comparison . . . 57
2.3.7 **RQ4:** Methods for Addressing Image-based Polyglots . . . 59
2.3.7.1 *ImSan*, a Content Disarmament and Reconstruction Tool for Image-based Polyglots . . . 59
2.3.8 Discussion . . . 60
2.3.8.1 Contribution Summary . . . 60
2.3.8.2 Limitations . . . 61
2.3.8.3 Future Work . . . 62
2.3.8.4 Recommendations . . . 63
2.3.8.4.1 Polyglot Detection . . . 63
2.3.8.4.2 Existing Signature-based Tools . . . 64
2.3.8.4.3 File-format Specifications . . . 64

3 Unguided Static Binary Instrumentation 67
3.1 Five Methods for Instrumenting PE files . . . 67
3.2 Instrumentation Variations . . . 70
3.3 Evaluation Scenario . . . 74
3.4 Results of Current Perturbations . . . 75
3.5 Implications of Current Research . . . 78
3.6 Future Perturbation Techniques . . . 80

4 Survey of Adversarial Binary Methods 84
4.1 Introduction . . . 84
4.1.1 Criteria and Contribution . . . 85
4.1.2 Survey Layout . . . 86
4.2 Background . . . 88

4.2.1 Definition of Adversarial Binary . . . 88
4.2.2 Domain Constraints of Binary Instrumentation . . . 89
4.2.3 Windows PE File Format . . . 90
4.2.4 General Malware Detection . . . 93
4.2.4.1 Dynamic Malware Detection . . . 94
4.2.4.2 Static Malware Detection . . . 95
4.2.5 Experiment Design Terminology . . . 96
4.2.5.1 Attack Scenarios . . . 97
4.2.5.2 Surrogate Models . . . 98
4.2.5.3 Transfer Property . . . 98
4.2.6 Optimization Algorithms . . . 99
4.2.6.1 Algorithm Selection Considerations . . . 99
4.2.6.2 Gradient-based Algorithms . . . 102
4.2.6.3 Genetic Algorithms . . . 106
4.2.6.4 Reinforcement Learning . . . 107
4.3 Surveyed Methods . . . 109
4.3.1 Target Models and Feature Sets . . . 109
4.3.2 Gradient, Genetic, and Other Guidance Mechanism Papers . . . 112
4.3.3 Algorithm Crossover Papers . . . 128
4.3.4 RL Guidance Papers . . . 130
4.3.4.1 RL Actions . . . 132
4.3.4.2 RL Algorithms . . . 135
4.3.5 Alteration Methods . . . 138
4.3.5.1 Taxonomy for Metrics and Methods . . . 139
4.3.5.2 Functionality Preservation . . . 140
4.3.5.3 Reinforcement Learning Actions . . . 144
4.3.5.4 Gradient-based Actions . . . 150
4.3.5.5 Genetic Algorithm and Other Guidance Mechanism Actions . . . 154

4.4 Conclusion . . . 158
4.4.1 Limitations . . . 158
4.4.2 Future Work . . . 160
4.4.2.1 Functional Malware Data Set . . . 160
4.4.2.2 Algorithm . . . 162
4.4.2.3 Target Models . . . 163
4.4.2.4 Feature Importance Recommendations . . . 164
4.4.2.5 Action Space . . . 165

5 Advancing Static Binary Instrumentation to Improve Evasion Rate 171
5.1 Introduction . . . 171
5.2 Explainability-driven Development . . . 173
5.2.1 Shapley Methods . . . 174
5.2.2 Shapley Results . . . 176
5.2.3 Novel Action: Section Virtual Negation . . . 178
5.3 Functionality Preservation . . . 180
5.3.1 Functionality Criteria . . . 180
5.3.2 Relevant PE Structures . . . 183
5.3.3 Evaluating Functionality Preservation of Prior Actions . . . 186
5.3.4 Novel Action Variants to Address Obfuscation . . . 192
5.4 Novel Action Effect on Evasion Rate . . . 198
5.5 Ablation Study . . . 200
5.5.1 Bound PGD Section Create at SoC Action . . . 206

6 Conclusion and Future Work 210
6.1 Conclusion . . . 210
6.1.1 Summary of Results . . . 210
6.1.2 Recommendations . . . 214
6.1.2.1 Evasion . . . 215
6.1.2.2 Detection . . . 216

6.2 Future Work 219

Appendix **238**

Vita **246**

Chapter 1

Introduction

Cybersecurity is, at its heart, a cat-and-mouse game between attackers and defenders. For every new defensive technique, an attack is soon developed that requires the development of a new mitigation. Signature-based detection of malware gave way to AI-driven detection as polymorphic and metamorphic viruses forced an evolution in detection strategies. AI-driven detection's ability to recognize novel malware is highly dependent on the similarity between training data and novel threats. This data dependency has driven the implementation of autonomous systems for producing novel malware that can be used to inoculate AI-driven detectors against emerging threats.

This body of work has largely drawn from the image classification domain where adversarial machine learning has been used extensively to improve the detection capabilities of image classifiers. In these methods, the gradient of a target or surrogate model is used to determine how to alter individual pixels in order to maximize the probability the image will be misclassified. In order to claim an altered image is mislabeled, it must have the same true label as the original image. Only then can we refer to the altered image as an adversarial sample. In order to maintain the true label, the alteration must be constrained in some way. This can be achieved by bounding

the number of pixels that can be altered or specifying some region of the image that must remain the same (e.g., the foreground object).

This restriction is not readily transferred to the malware domain due to two factors. First, each byte that is used to perform an action (opcode) or store a variable (data) may necessary for the proper execution of the file. Alteration of a single one of these bytes could prevent the file from executing altogether. Second, loss of information in the compilation process of some executable formats—especially Windows PE files that rely on the most common architectures, X86 and X86/64—means that we can not determine with 100% confidence the role of every byte in the file. We focus on the Windows Portable Executable format due to its ubiquity and difficulty. Variable names are stripped away; all structures and objects are flattened into arrays; data length and, therefore, type is lost as well. Additionally, the Windows PE file must inform the program loader how to allocate resources before the file can be loaded into a waiting process in virtual memory. This mapping from raw bytes to virtual addresses introduces another layer of complexity that must be respected. Any alteration to the header of the PE file could prevent the file from being mapped into memory. Any change in the executable code could prevent the file from executing its intended behavior. Any change in the read/write sections could overwrite a necessary variable and prevent execution. Due to this myriad of concerns, altering a PE file is like walking through a mine field.

We know there are safe-to-alter bytes within the various regions of the file. Some of them are visible simply by parsing the headers. Others, however, are buried by the compilation process, forcing us to either avoid those byte regions altogether or sift through their contents using conservative disassembly via a reverse engineering tool. At all times, we must consider that any change will interfere with functionality. By the definition of adversarial sample given by Szegedy Szegedy et al. (2013) and Demetrio Demetrio et al. (2021c), functionality verification is a critical step when crafting adversarial malware samples, which we refer to henceforth as adversarial binaries. If an altered file can no longer exhibit malicious behavior, then it can not be

termed an adversarial binary. This dissertation is focused on the creation of adversarial binaries.

One may rightly wonder why we need to create adversarial binaries in the input space instead of generating adversarial features without attempting to map back to the input space. We do not disregard this approach that has proven effective in the image domain. However, we point out that the feature extraction process for malware is highly variable. Extracted features may be static or behavioral. Feature sets may include the raw bytes of the file (up to some maximum capacity), extracted and compressed fields from the header, or some transformation of a subset of the file. A control flow graph, for example, can be extracted using a reverse engineering tool like Ghidra or IDA Pro for training a graph neural network. Alternatively, opcodes could be extracted for N-gram analysis using a natural language processing (NLP) approach. Due to this variation in feature sets, generating adversarial feature vectors to provide novel training data faces a severe restriction.

Adversarial feature vectors can only be used by models that share the same exact feature space. Since industry is strongly motivated to keep their detection methods secret, little is known of the exact features used by leading vendors. This limits how widely researchers can provide novel training data. Adversarial binaries, however, can be shared in a feature-agnostic manner. A researcher does not need to know which features will be extracted from the binary in advance in order to provide useful training data. It should be mentioned that a vendor may not extract the same vulnerable features that a particular researcher targeted; however, this test would still provide useful information to the vendor, allowing them to internally quantify their vulnerability or lack thereof to that particular type of adversarial sample. As Chapter 2 will demonstrate, there are also practical attacks on feature extraction that mean the extracted features may not match the input. If all attacks are modeled in the feature space rather than the input space, there is no opportunity to asses the effect of evading feature extraction. We argue that, in order to secure AI-driven malware

detectors to the best degree possible, we must model input space threats, i.e., create adversarial binaries.

1.1 Contribution: Chapter 2

This chapter addresses feature-extraction evasion and provides real-world justification for modeling threats in the input space (actual binaries) rather than feature space. ORNL conducted an enterprise-scale analysis of four leading commercial off-the-shelf malware detectors for the U.S. Navy. As part of this experiment, Assured Information Security (AIS) provided a set of 1024 malware samples designed to mimic the efforts of a sophisticated opponent. All 1024 samples evaded detection by all four detectors, prompting further investigation. These files were *polyglots* composed of a JAR file combined with a JPEG image.

A polyglot is any file that simultaneously conforms to two or more format specifications. A polyglot's behavior is dependent upon the calling program. In other words, a JPEG+JAR polyglot will present an image when parsed by an image viewer but execute malicious self-extracting Java code when parsed by the Java Runtime Environment. While our agreement with COTS vendors prevented us from investigating the cause of the 100% failure rate, we can theorize that the abnormally high evasion rate was due to a failure to identify the JAR component of each polyglot. We note here that malware detectors, in an attempt to extract the most discriminative features, are often format-specific. Ergo, they first identify the format of the file, then pass the file to a submodel or subset of signatures specific to that file type. This means that format identification plays a key role in feature extraction. If the JAR component was not extracted or compared to JAR signatures, then the detection models applied their label based on the benign JPEG contents. This could explain the unusual 100% evasion rate.

Based on this test of COTs tools, we began a project to generate, detect, and remediate polyglot files. We note that Windows PE files can be formed into polyglots

provided that certain offsets are respected. In this case, turning a Windows PE file into a polyglot can be considered a special type of *Overlay Append* or *Section Append* action, depending on the combination method. This places polyglots into the context of static binary instrumentation methods for AI-driven detection evasion. Polyglot files pose a particularly unique threat compared to other binary instrumentation methods due to their ability to evade correct feature extraction entirely.

We focused on detection and remediation to provide a drop-in solution for existing malware detectors. Rather than retrain or rebuild their entire malware processing pipeline, vendors need only use our tool to detect polyglots prior to format-specific routing. The polyglot can then either be quarantined or passed to all matching format-specific detectors rather than only one detector. This project consisted of two papers.

For the first paper, we generated a balanced dataset using the open-source Mitra tool and evaluated a variety of feature sets and model architectures. We compared these features and models to existing tools for file identification, polyglot detection, and file carving. Our top-scoring model was a CatBoost model trained on byte unigrams concatenated with the output of *file*. However, we wondered if a larger dataset would yield improved results for the deep learning model we tested.

For the second paper, we first conducted a survey of polyglot usage by malware authors (to our knowledge, the first of its kind). We found that these files have been actively used by malware authors and play a critical role in bypassing certain defenses for some high-profile malicious campaigns. Based on the survey results, we built a much larger dataset than our previous effort and modeled it on the formats and combination methods used by malicious actors in the wild. Next, evaluated the top-scoring models from the workshop paper, finding that deep learning was the optimal approach once the scale of the dataset was sufficient. We compared this model to open-source file format identification tools—including two that were polyglot-aware—and found our model outperformed them by a large margin at both the binary task of identifying if a file was a polyglot and the multi-label task of identifying all the formats contained

in a file. Lastly, we built a content disarmament and reconstruction (CDR) tool that sanitized image-based polyglots.

In sum, the polyglot work resulted in a published paper Koch et al. (2022), a second paper in the publishing process Koch et al. (2024), and two technology transfer agreements with anonymous organizations. With regard to this work, the use of polyglot files in real-world campaigns highlights the importance of creating and testing evasive samples in the input space rather than relying on attacks enacted only in the feature space. Having provided solutions for feature extraction evasion, we focused on altering input files so that the correctly extracted features evade detection.

1.2 Contribution: Chapter 3

This set of experiments with static binary instrumentation was part of a larger project to build an automated system that would generate adversarial binaries. The system, termed the Adversarial Malware Input GeneratOr (AMIGO), was conceived as a four stage modular process. A file would be passed through a surrogate model and the significant features extracted using saliency vectors. Guided by these vectors, a binary instrumentation method would be applied to alter the file. A verification step would then be applied to confirm the altered file's functionality. Finally, the altered file would be presented to the target detector to determine if it succeeded in bypassing detection. If it succeeded, the file would be labeled an adversarial binary and added to a data set. This data set could then be provided to vendors and researchers to inoculate their AI-driven detectors.

The component of this system that was completed and tested by ORNL was the perturbation library, a set of static binary instrumentation methods for altering Windows PE files. This set of instrumentation actions—sans guidance from any mechanism—was tested against two commercial detectors in use by a sponsor organization. One system was signature-based while the other was AI-driven. The

results showed a high evasion rate for actions that altered the structure of the PE file, especially the section table.

These results prompted an interest in the structure of PE files and how these fields–which are used to map the file into a process—could be manipulated without loss of function. This interest led to a study of the Windows PE file's format specification. The format, while more complex than most, tolerates certain deviations from the format specification that can only be revealed via experimentation or the study of malformed malware files that can still execute.

The AMIGO system was copyrighted by ORNL, licensed to a startup, and was recently granted a patent (U.S. Patent No. 12,019,746).

1.3 Contribution: Chapter 4

Having demonstrated that input-space perturbations are real threat to malware detection systems, we turned our attention to optimization methods for these perturbations. This chapter relates our survey of adversarial binary creation methods. The scope of the survey covered papers that have combined an AI-guidance mechanism (gradient, genetic, or reinforcement learning) with one or more static binary instrumentation methods to create adversarial Windows PE binaries. In the survey, we discussed the advantages and disadvantages of each family of guidance algorithm that has been applied to the adversarial binary generation problem. We also provided a summary of prior instrumentation actions to facilitate future development of more complex actions.

Five particular shortcomings emerged from this survey of prior work. First, assumptions that functionality will be preserved by actions that nominally follow the PE format specification have been proven invalid. Second, functionality verification criteria are vague and some of the tools used as an oracle for functionality preservation were designed specifically to parse PE files despite malformation. We highlighted three papers that more thoroughly tested functionality and used the strictest combination

of their criteria to evaluate our work. Third, we noted that none of these papers have conducted an analysis of the obfuscations which may violate format specifications found in some malware files. Since these obfuscations may interfere with functionality, identifying them is the first step to developing instrumentation actions that preserve functionality of heavily obfuscated malware. Fourth, we noted that—with one recent exception—papers which relied on reinforcement learning did not optimize the content utilized in their corresponding actions, which means that the evasion rate reported by these papers might be improved via content optimization. Fifth, we note that only two papers used a feature importance metric to determine which features to target or prioritize. By first conducting this analysis, researchers could avoid investing significant time in developing a functionality preserving action that does not significantly affect the features of a target model. These shortcomings largely inspired the two experiments which are detailed in the final chapter. This survey is currently under review for publication.

1.4 Contribution: Chapter 5

The survey revealed that the development of novel binary instrumentation actions appears to be a limiting and unique factor for research in this domain. Although much prior work focused on implementing ever more complex algorithms for guiding instrumentation, simpler agents performed well when paired with more effective actions. We therefore designed an experiment to confirm that adding more effective actions while holding all other experiment parameters constant would yield a more effective adversarial agent. Given the large space of possible binary instrumentation actions, we took a principled approach using feature importance metrics and surrogate models to find actions that increase the evasion rate for adversarial agents.

We first explored the feature importance method using Shapley values. Based on the Shapley values, we created a novel action, *Section Virtual Negation*, and tested its effectiveness both as a stand-alone action and as part of an RL agent's action

set. We compared the RL agent's evasion rate before and after adding this action, finding that our action was selected often by the agent and increased the evasion rate in both soft-label and hard-label agents. Having completed our main experiment with positive results, we conducted an ablation study to determine which factors most contributed to the improvement in evasion rate. We found that the Shapley values were not reliable, but that the byte and byte-entropy histograms were a driving factor for evasion.

An interesting artifact that emerged from the ablation study was an usual correlation between the benign label and random data appended to the overlay, but not random data in a section. Based on our polyglot research, we speculate that benign self-extracting Zips were present in the EMBER data used to train the detector. Note, the term EMBER refers to both a data set and a feature set Anderson and Roth (2018a). We have reached out to the EMBER authors for confirmation.

The ablation study prompted a second experiment using a surrogate model instead of feature importance. The surrogate model method paired a convolutional neural network (CNN) model trained on EMBER features with a bound and masked version of the projected gradient descent (PGD) attack. We used the bounds and mask to target the byte histogram, and produced a reconstruction method for mapping the PGA attack back to the input space via an *Overlay Append* action. This action appends data to the end of a PE file without mapping it into a section; it was highly effective against the EMBER gradient-boosted decision tree (GBDT) target model. We then searched the literature for similar attacks, finding one that used a generative adversarial network (GAN) to target the byte histogram. This method, however, generated a much lower evasion rate, albeit on a different malware dataset. Given the high evasion rate, we sought to reach 100% evasion by improving the action further. Using our knowledge from the last experiment, we added the byte histogram data to a section and placed it early in the file (*Bound PGD Histogram Section Create* action) in order to stymie heuristic detection of the attack, a clear limitation of appending to the overlay. By placing the data in a section, we were able to reach 100% evasion

on our functionality testing subset, and 97% on a larger random sample of malware while also preventing heuristics from detecting unmapped data in the overlay.

As highlighted in the survey, binary instrumentation that is not functionality preserving contradicts the definition of an adversarial binary. Therefore, thorough criteria for functionality testing are a key component of novel action development. To this end, we developed and adhered to the strictest criteria yet published for verifying functionality. We highlight the importance of functionality verification prior to alteration, finding a high degree of variability and anti-sandbox behavior in our functionality testing subset that would have interfered with our results if not detected. We cannot detect loss of function in a file that exhibits no functionality prior to alteration. In other words, prior results that claim functionality preservation without asserting detectable functionality prior to instrumentation are suspect. Furthermore, we are the first to highlight specific ways that malware files deviate from file-format specifications by examining difficult-to-instrument malware. Based on this examination, we developed more advanced instrumentation methods to perturb these deviant files without loss of function according to our strict criteria. Our development history is provided so that future researchers do need to tread the same ground in terms of experimentally verifying which components of the file-format specification must be respected and which can be ignored.

Chapter 2

Polyglot Creation and Detection

2.1 Contribution Summary

Methods for obfuscating a Windows PE file are not limited to changes within the PE file's structure. A PE file may be obfuscated by combining it with another file into a *polyglot*. A polyglot is a single file whose contents conform to multiple file format specifications simultaneously. In other words, a PE+HTA polyglot is both a valid PE file and a valid HTA file. Whether the PE contents or the HTA contents are executed depends on the calling program, which varies depending on user interaction or OS-specific auto-launch behavior. Polyglot files pose a threat to format-specific malware detectors that rely on file-format identification to select which submodel or subset of signatures will analyze the malware. In other words, polyglots are a method for evading feature extraction that is unique to the malware domain. Before we could consider the threat posed by manipulating input to evade classification we first needed to ensure that the extracted features represented the correct content.

In this chapter, we investigated the threat posed by polyglot malware and developed tools for detection/remediation. The work is divided into two papers. The first paper used a balanced small-scale dataset to determine which model architectures and feature sets were most promising. The second paper presented a survey of polyglot usage

in the wild, a much larger dataset based upon malicious polyglot usage in the wild, binary and multi-label detectors, a content disarmament and reconstruction (CDR) tool, and YARA rules. Although the bulk of this dissertation is focused on finding vulnerabilities, this chapter explores potential defenses against fueature extraction. Chen argued that adversarial retraining, although widely used, has the potential to poison a malware detector if too many samples are used for retraining Chen et al. (2019). Instead, Chen argues for filtering out adversarial malware. This chapter follows a similar vein, first illustrating the danger posed by polyglots and then providing detection and remediation tools.

2.2 Toward the Detection of Polyglot Files

2.2.1 Introduction

2.2.1.1 Overview

A polyglot is a form of steganographic file that can be successfully interpreted as two or more types Albertini (2015); Shah (2015); Ortiz (2019). In other words, a JPG+JAR polyglot is one that presents an image when interpreted by an image viewer and executes self-extracting Java code when fed to the Java Runtime Environment. Many combinations are possible, including a polyglot that can be interpreted as an image, a video, a PDF, and a video game depending on which program interprets the polyglot Albertini (2015). As we demonstrate in our results (section 2.2.4), common utilities for identifying files do not reliably identify polyglots. Ergo, malware detection systems that rely on common utilities for file type identification prior to feature extraction are vulnerable to polyglot files. If a detector recognizes only the JPG portion of a JPG+JAR polyglot, then two problems can arise.

- For machine or deep learning-based detectors, feature extraction will only extract fields from the JPG portion of the file

- For signature-based detectors, only malware signatures associated with JPG files will be matched Jana and Shmatikov (2012a)

In either instance, malicious code in the JAR portion passes through without accurate analysis. In 2019, Bridges et al.Bridges et al. (2020) analyzed commercial malware detection systems and found that multiple systems failed to detect polyglot malware. Motivated by this failure, we compiled a data set of normal (which we refer to as monoglot) and polyglot files, trained multiple classifiers on this data set, and evaluated the recall, precision and F1 score of these classifiers. Our objective is to protect malware detection systems that rely on file type identification for feature extraction or signature sub-selection by developing a model to filter polyglots out of the file processing pipeline. This will allow commercial tools to filter out potentially malicious polyglots while still benefiting from the rich information,e.g. header metadata, provided by file type-specific feature extraction Anderson and Roth (2018b).

2.2.1.2 Motivation

Malware detection systems commonly rely on file type-specific features in order to classify an unknown file as either malicious or benign Anderson and Roth (2018b). For example, the header of a PE file contains a great deal of information about the file, e.g., the read/write/execute flags of all sections, the offset to the import function table, and the address of the entry point. In a PDF file the XREF table, if present, is a useful source of information for determining which portions of the PDF could contain malicious activity. If the XREF is not present or consistent with the file contents, then malicious code can be found by scanning for *action* objects that launch Javascript code Maiorca et al. (2019); Popescu (2012b). In any event, knowing the file type allows the detector to extract far more information than otherwise possible. If the file type is not known, then only generic features like byte entropy, n-grams, and strings can be used for classification Anderson and Roth (2018b).

In some instances, malware detectors will ignore files if the file type has no known malicious capability. If a polyglot file is identified as having only one file type, only that type will be used for feature extraction and further analysis. File sections from the undiscovered type will simply be ignored, allowing potential malicious activity to go undetected or misunderstood Jana and Shmatikov (2012a). One solution is to forego file type-specific feature extraction. However, this greatly reduces the number of available manual features Anderson and Roth (2018b); Saxe and Berlin (2015). Deep learning (DL) could be used to perform automatic feature extraction that is effective across all desired file types. However, this would require the complete overhaul of a file type-specific malware detection system. Instead, we propose simply adding a polyglot filter as a pre-processing step for existing malware detection systems. Provided the polyglot filter has sufficiently high recall, this solution would require minimal alteration to existing solutions. Furthermore, existing solutions would continue to benefit from the rich features that can be extracted once a file is known to contain only one file type.

Additional evidence of the pressing need for polyglot detection is provided by a 2019 attack on DICOM files Desjardins et al. (2020); Sundareswaran and Squicciarini (2010). DICOM files are commonly used medical imaging files. The attack involves the creation of a PE+DICOM polyglot that functions as expected when loaded into medical imaging software, yet is also capable of execution as a Windows PE file. The DICOM type is intentionally flexible; the authors anticipated the creation of TIFF+DICOM polyglots for the convenient bundling of medical scan images Graham et al. (2005). However, this flexibility lends itself well to more malicious pursuits when the second file contains malicious code. Exactly how this flexibility can be exploited is described in the related work (section 2.2.2.2).

Due to the wide variety of file types that exist, we felt that a signature-based approach to polyglot detection would require excessive revision and, at best, only account for previously-encountered forms of polyglots. There are also a large number

of discrepancies in terms of malformed files that make the signature-based approach difficult Jana and Shmatikov (2012a).

In an attempt to create a solution that learns the general problem and has the possibility of protecting against novel polyglots, we decided to explore machine and deep learning for binary polyglot detection. In contrast to normal file type detection tools where the goal is to identify the correct file/mime type, the goal of our models is the binary classification of a file as polyglot or monoglot (non-polyglot single file type). This required the creation of a sizable training set that includes a wide variety of both monoglots and polyglots. Before we discuss this data set, we first provide a detailed look at the various types of polyglots that are included within it.

2.2.2 Background

2.2.2.1 Polyglot Mechanics

Although there has been some academic research into file type identification methods (discussed in section 2.2.2.2), polyglot creation has largely been driven by researchers from industry Albertini (2015); Shah (2015). Ange Albertini demonstrated multiple methods for creating polyglots at industry presentations and released the `python`-native `mitra` tool on Github so fellow researchers could generate polyglots. We utilized this tool to create our polyglot data set. Albertini classified polyglots based on the method used to combine donor files into the polyglot. Ergo, `mitra` attempts to create stacks, zippers, parasites, and cavities from pairs of donor files.

2.2.2.1.1 Stacks

A stack is the simplest type of polyglot. The second file is simply appended to the end of the first file. The caveat is that any byte offsets in the second file may need to be adjusted (increased by the length of the first file) in order for the second file to function as expected. The first file must tolerate appended data while the second file must not strictly enforce the *magic number at offset zero* rule. A **magic number** is

an arbitrary (hence the *magic* moniker) hexadecimal value that uniquely identifies a file type. This number is used for fast file type identification, since a utility need only scan the first few bytes of the file in order to identify it. However, there are many file types that do not enforce this rule.

2.2.2.1.2 Parasites

In a parasite polyglot, the second file is added within comment sections—offset by comment markers—of the first file. Many file types allow for comment sections that are not displayed when the file is interpreted. These comments are only visible when the file is opened for editing by a hex editor like `vim`. In order for the second file within the comment sections of the first file to remain functional, it must not have the strict *magic number at offset zero* rule. The byte offsets for both files must be updated. In the case of the first file, the byte offsets must now account for the second file contents that are now hidden within the comment sections. In the case of the second file, the offsets must account for the fact that the second file contents are now scattered in comment sections of the first file. As long as these offsets are updated correctly, both files will continue to function normally. Although this update process is more complex than the updates necessary for a stack polyglot, there are many file types than can combine to create a parasite.

2.2.2.1.3 Zippers

Zippers are a more complex version of a parasite. In a zipper, both files are contained within each other's comment sections. This means that the two donor files must use different markers to begin and end their comment sections. This arrangement is rather unusual in practice, so donor files for zippers are much harder to find. In our data set, only DCM files combined with either GIF or PDF files were able to create zipper polyglots.

2.2.2.1.4 Cavities

Cavities are created when the second file is hidden within null-padded areas of the first file. This arrangement is only possible when the first file is of an executable or ISO type, wherein memory is allocated in chunks. Since these areas are often null-padded to a standard size, executable or ISO files may contain enough null-padded memory to hide a second file in the padded areas. This is very similar to the classic *code caving* technique often used by malware authors to hide malicious code. The distinction between a cavity polyglot and code caving is that the caved material of a cavity polyglot is a complete file that can function correctly when interpreted by an appropriate program. Since the first 16 sectors of an ISO file are left empty, the contents of another file can be placed in the beginning of the ISO file. For executable files, like Windows PE files, the second file would be written into the null-padded trailing areas at the end of sections. No updates will be needed for the first file, but any byte offsets in the second file will need to be updated if the second file does not begin at offset zero of the first file.

2.2.2.2 Related Work

2.2.2.2.1 Related Tools

File type identification has historically been addressed through signatures Darwin et al. (2019). The Linux utility `file` matches files by examining magic bytes in unison with other structural elements. The magic number is an arbitrary hexadecimal value that uniquely identifies a file type. Some file types can be strictly identified simply by their magic number, while others require additional elements to be scanned Darwin et al. (2019). JAR files, for instance, are simply Zip files with a MANIFEST.INF file present in the archive. Ergo, `file` scans for the presence of this file in order to distinguish a JAR file from a Zip file. In either event, `file` matches each input file to a unique signature. Note, `file` scans until it detects a match, then halts. There is a `--keep-going` flag, but in our examination it did not allow `file` to detect polyglots.

The results of running `file` (with this flag active) on our polyglot data set is included in section 2.2.4.

`binwalk`[*] is an open-source file carving tool routinely used by reverse engineers and malware analysts to extract hidden code. When fed a file, `binwalk` outputs all the file/data types that it recognizes within the file. `TrID`[†] is a file format identification tool currently in use by Virustotal[‡]. It is very similar in function to `file`. Lastly, `polydet`[§] is an open-source polyglot detection tool.

We are also aware of `polyfile`[¶], which is an open source tool that uses a cleanroom implementation of libmagic and recursively searches the fields in a file looking for multiple file types. When we initially tested `polyfile` it performed poorly enough that we left it out of the evaluation, but due to recent modifications we are retesting it. Our current results suggest that while it has improved it does not significantly outperform other existing solutions and gives results similar to a combination of `file` and `polydet`.

2.2.2.2.2 Related Research

McDaniels and Heydari used byte histograms in concert with three different algorithms to identify file types McDaniel and Heydari (2003). Under the first algorithm (*byte frequency analysis* or BFA), they converted files into a fixed length feature vector. All files, regardless of type, can be represented as a sequence of hexadecimal values. Therefore, we can compactly summarize a file's contents by placing the number of times each possible hex value occurs into a 256 (all possible hexadecimal values) character vector where the index corresponds to the byte value. Ergo, the 7th value stored in the vector is the number of times the value 0x7 occurred in the file. This feature is referred to as a byte occurrence vector, unigram, or byte histogram in literature. McDaniels and Heydari then normalized each byte occurrence vector and

[*] https://github.com/ReFirmLabs/binwalk
[†] https://mark0.net/soft-trid-e.html
[‡] https://virustotal.com
[§] https://github.com/Polydet/polydet/tree/master/polydet
[¶] https://github.com/trailofbits/polyfile

calculated an average vector for each file type. Each average vector represented a unique file type and was referred to as a *fileprint*. Test files were compared to the *fileprint* to calculate correlation scores. Finally, each test file received its file type label from the most correlated *fileprint*.

Karresand Karresand and Shahmehri (2006), Veenman Veenman (2007), Fitzgerald Fitzgerald et al. (2012), and Beebe Beebe et al. (2013a, 2016) all trained machine learning (ML) classifiers that used a wide variety of statistical features extracted from files and fragments, which included unigrams, bigrams, bag-of-words, byte rate-of-change, Kolmogorov complexity, and common/longest strings/bytes. Models developed included support vector machines, K-nearest neighbors, and hierarchical clustering. Beebe released an open-source tool, Sceadan, which achieved 73.7% accuracy on 30 file types and 8 data types by utilizing a linear SVM trained on an input vector of unigrams concatenated with bigrams. Unigrams simply the natural language processing (NLP) term for the byte occurrence count vector or byte histogram. Bigrams are a 256x256 matrix of co-occurrence counts. This data is usually sparse and slow to compute Mittal et al. (2021).

More recently, deep learning has been used to detect the type of file fragments. File fragments may contain a partial signature or no signature at all, making rule-based approaches useless. Moreover, deep learning has an advantage over other learning algorithms thanks to its automatic feature extraction, which allows deep models to train on raw bytes. Mittal et al. built a deep learning model, FiFTY, to identify file fragments in 2021 Mittal et al. (2021). They trained a one-dimensional convolutional neural network to identify 75 different file types. Although the authors experimented with training neural networks and convolutional neural networks on a vector of statistical features that included Shannon entropy, Kolmogorov complexity, deviation, skewness, kurtosis, and mean values (arithmetic, geometric, harmonic) as features, raw bytes as the only input proved the most accurate. Their model outscored Sceadan (77.5% acc vs 69.0% acc) on a data set of 75 file types, which was released along with the FiFTY model.

In terms of cybersecurity, polyglot files are the subject of a modest number of publications Desjardins et al. (2020); Sundareswaran and Squicciarini (2010); Jana and Shmatikov (2012a); Brandis (2009); Dhanjani et al. (2009). Jana and Shmatikov detailed a wide variety of obfuscation methods that target discrepancies between how a malware detector parses a file and how the OS interprets the file Jana and Shmatikov (2012a). Some of these discrepancies cause the detector to misidentify the file type of the incoming file, resulting in the wrong subset of malicious signatures being applied to the file. These methods are referred to as Chameleon attacks Jana and Shmatikov (2012a). Werewolf attacks Jana and Shmatikov (2012a), on the other had, exploit discrepancies in how the file is parsed. Jana and Shmatikov describe one type of Werewolf attack as 'ambiguous files conforming to multiple types'Jana and Shmatikov (2012a). These are polyglot files by another name. The authors note that a malicious TAR+ZIP stack-type polyglot was incorrectly classified by 20 out of 36 malware detectors hosted on Virustotal at that time.

Nitesh, Hardin, and Rios examine the browser-based GIFAR (GIF+JAR) polyglot Dhanjani et al. (2009). This polyglot is specifically designed to evade file type-based protections built into browsers by smuggling the JAR component in behind the GIF component. Brandis also discusses this type of polyglot in his white paper Brandis (2009).

Desjardins et al. report on PE+DICOM polyglots and the threat they pose to medical records Desjardins et al. (2020), also discussed by Selvaganapathy and Sadasivam Sundareswaran and Squicciarini (2010). This polyglot presents as normal medical images when opened in an image viewer, but is also capable of launching its executable component when run from the Windows command prompt. In this case, the PE header is the first element in the file followed by the entirety of the DICOM data. The rest of the PE file follows the DICOM data. The PE header is ignored by DICOM imaging software which simply reads the DICOM components. DICOM files have an optional preamble designed to boost compatibility with non-DICOM image viewers; therefore, DICOM viewers treat the PE header as an optional preamble that

is meant to be ignored. The PE header, meanwhile, points past the DICOM data to the beginning of the PE contents, allowing the executable content to run without error when launched from the command prompt.

2.2.2.3 Relation to This Work

Our key contribution is to show that machine learning models have strong potential for polyglot detection, while existing file type analysis based on semantic structure and/or libmagic fails. Our work builds upon three of the methods presented above. Our data set consists of polyglot files whose basic structure was described by Jana and Shmatikov, albeit with a different nomenclature. Courtesy of Mittal Mittal et al. (2021), we train a one-dimensional convolutional neural network on raw bytes. We also train a variety of ML models on the byte count histogram as detailed by McDaniels and Heydari McDaniel and Heydari (2003).

2.2.3 Methodology

2.2.3.1 Polyglot Data Set

Our dataset, described in Table 2.1, consists of 10 monoglot or normal files types as the negative class and 21 polyglot file types as the positive class with an 80/20 train/test split. We should note that these polyglots do not contain malicious code. Our objective is a polyglot filter, not a malware detector. Additionally, the use of benign polyglots greatly simplifies data sharing.

There are 3120 of each file type in the monoglot training set and 780 of each file type in the test set. The monoglot file types are as follows: PDF, PNG, GIF, TIFF, JPG, DCM, JAR, Zip, PE, and ISO. Feeding a subset of the above files to `mitra`, we created 21 different polyglot combinations. There are 1200 of each file type combination in the training set and 300 in the test set.

The files used to create our dataset were scraped from the web between June 8, 2021 and November 19, 2021. We scraped files from archive.org (ISO), arxiv.org

(PDF), purepng.com (PNG), imgflip.com and forgifs.com (GIF), kaboompics.com (JPG), kaggle.com (DCM, TIFF, GIF, PNG, JPG), and sourceforge.net (ZIP, PE, JAR). Each file type was split into three size categories (small, medium, and large) relative to the smallest and largest files collected for that file type. An equal number of each size category was used to generate polyglots.

Since each pair of file types can only be combined in certain ways (stack, zipper, parasite, cavity), the number of each type of polyglot in this data set is not balanced. Our future work includes the production of a data set that is balanced according to polyglot combination method rather than file type. Table 2.1 contains the breakdown of polyglot combination methods in our training and test sets.

2.2.3.2 Model Selection

We trained six types of model to identify polyglot files: deep neural network, random forest, CatBoost, GradientBoost, LightGBM, and support vector classifier. Traditionally, machine learning models used in cybersecurity applications rely on manual features that are specific to each file type Souri and Hosseini (2018); Merabet and Hajraoui (2019). Since polyglot files can confound file type-specific feature extraction, our models are file type agnostic. In the case of ML models trained through `scikit-learn`, the input vector consists of a byte frequency analysis, also known as a byte histogram or unigram, as described in Section 2.2.2.2. Since files are stored internally as hexadecimal values (hence 256 possible values for each byte), this feature can be used regardless of file type.

Our second method concatenated the byte histogram vector with the mime-type output of the utility `file`. Since `file` is very accurate on normal files, we theorized that adding `file`'s output to the feature vector would lessen the learning load on a model.

We also trained and tested a deep learning model. Deep learning via a convolutional neural network (CNN) utilizes automatic feature extraction Raff and Nicholas (2020); Raff et al. (2018a, 2020a), making it ideally suited for our task. The deep learning

model we chose is designed for binary (two class) classification of binary (compiled) files, namely, MalConv2 Raff et al. (2020a). This model is a one-dimensional convolutional neural network developed for malware detection whose feature vector consists of the raw bytes. The first MalConv model demonstrated that CNNs can be effective malware classifiers Raff et al. (2018a). However, it had some issues that prompted the authors to develop a second, more effective model.

Since MalConv read in raw bytes as features, the input file had to be truncated if it exceeded the maximum capacity of the model (2MB). In the second iteration of the model, MalConv2, the authors exploited the sparse nature of temporal max pooling Raff et al. (2020a). This allowed them to partition the varying size files into N sections, then collect only the bytes that would actually get updated with nonzero gradient during the backward pass of the training process. This temporal max pooling increased the model's capacity from 2MB to 16MB. Moreover, Malconv2 features a gating mechanism that allows the model to correlate features that are distant from one another in the byte space Raff et al. (2020a). This is especially important for polyglots, where two separate headers might be in the same file at a great distance from one another.

We note that one criticism of MalConv is that it pays far more attention to headers than data or code Demetrio et al. (2019a). Although this was an undesirable trait in a malware detector, it is useful for a polyglot detector since the header structure is closely tied to the file type. Note, some file types utilizes footers instead of headers or have no distinct header/footer, so learning which parts of a file are most relevant is a challenging problem for a polyglot detector.

In cybersecurity a false negative means that a malicious file is allowed onto a host system. Ergo, we consider recall the most important statistic. Secondarily, a large number of false positives can overwhelm a security operations center (SOC); therefore, we report precision as well. Since the F1 score balances precision and recall in one statistic, we also pay attention to it. Our winning models are those with a high recall

and high F1 score. Note, all of the figures in the results section refer to the test set described in Section 2.2.3.1.

2.2.3.3 Train/Test Split and Hyperparameter Tuning

For random forest, CatBoost, GradientBoost, LightGBM, and the support vector classifier we used an 80/20 train/test split. We then did additional hyperparameter tuning for the two top performing models, which were CatBoost and random forest. Scikit-learn's GridSearchCV function was used to search the hyperparameter space with three-fold cross validation. For CatBoost, we passed the following hyperparameters: `depth`=10, `iterations`= 1000, `l2_leaf_reg`= 1, `learning_rate`= 0.15. And for random forest: `n_estimators`= 1400, `min_samples_split`= 2, `min_samples_leaf`= 1, `max_features`= 'auto', `max_depth`= 10, `bootstrap`= `False`. For CatBoost and random forest, hyperparameter tuning provided an approximately 1-2% increase in performance.

For Malconv we also used the 80/20 train/test split. However, with Malconv we used the default hyperparameter settings because attempts at manually tuning hyperparameters did not improve performance. It is possible that there is room for improvement in the hyperparameters for Malconv that could be explored in future work.

2.2.4 Results & Discussion

2.2.4.1 Testing Existing Tools

We begin our results with an examination of how well existing file type detection tools detect the multiple file types within polyglot files. Note that `file`, `binwalk`, and `TrID` are not designed to detect polyglot files, while `polydet` was written specifically for polyglot detection (see Section 2.2.2.2.1 for descriptions of each solution). We evaluate the performance of `file`, `binwalk`, and `TrID` on polyglots because these tools are routinely used to fingerprint incoming files for further analysis and therefore represent

the current level of industry vulnerability to polyglot files. Firstly, Figure 2.1 shows that the most common method for identifying file types, `file`, has poor overall recall on the test set of monoglot and polyglot files. `file` does perform extremely well on monoglot (normal) files, as demonstrated by its high precision. Ergo, `file` is highly effective on expected input, but fails on polyglot files. We used the `-keep-going` flag when running `file` to give `file` the best chance to detect both file types in a polyglot. Along with `file`, we tested the three other tools mentioned in Section 2.2.2.2: `binwalk`, `TrID`, and `polydet`. As Figure 2.1 demonstrates, none of these tools achieves high recall or F1 score.

Figure 2.2 breaks down `file`'s performance by polyglot type. `file` does perform well on zippers and cavities; however, the requirement for mutual comment markers and padding space, respectively, means that only a handful of file formats can produce these polyglots. Ergo, they constitute only 18.83% of the test set (see Figure 2.3 for a break-down of file combinations for each polyglot type). `file` performs poorly at detecting the more common stacks and parasites. The key takeaway from this evaluation of existing file type detection tools is that detecting polyglots is a non-trivial task and that there is no simple combination of existing solutions that can solve the binary polyglot detection problem.

2.2.4.2 Testing Machine Learning and Deep Learning

Next, we examine the efficacy of machine learning and deep learning for the binary polyglot detection problem. Figure 2.4 shows the precision, recall and F1 score of five machine learning models and one deep learning model (MalConv2). The ML models are trained on the byte histogram of each file while the deep learning model is trained on the raw bytes.

Figure 2.5 details MalConv2's performance on each polyglot combination method in the test set. Unlike `file`, MalConv2 has high recall for all polyglot types. That said, precision is lower for zippers and cavities.

Table 2.1: Data Set Composition

Data Set	Train	Test
Monoglot	31,199	7,799
Polyglot	25,210	6,303

Polyglot Type	Train	Test
Stack	10,267	2,574
Parasite	10,301	2,542
Zipper	1,795	463
Cavity	2,847	724

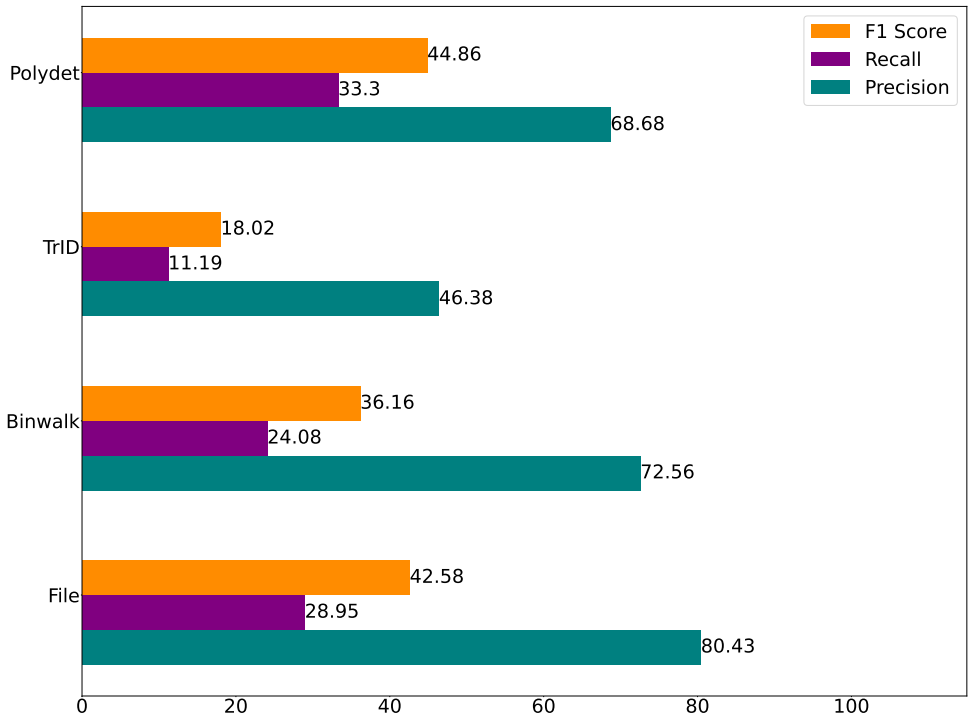


Figure 2.1: Existing tools for file type identification/extraction fail to reliably detect polyglot files. Although `file` displays relatively high precision, we consider this less important than recall and F1 score in the context of cybersecurity. `polydet` has the highest recall and F1 score, yet performs far below an acceptable level.

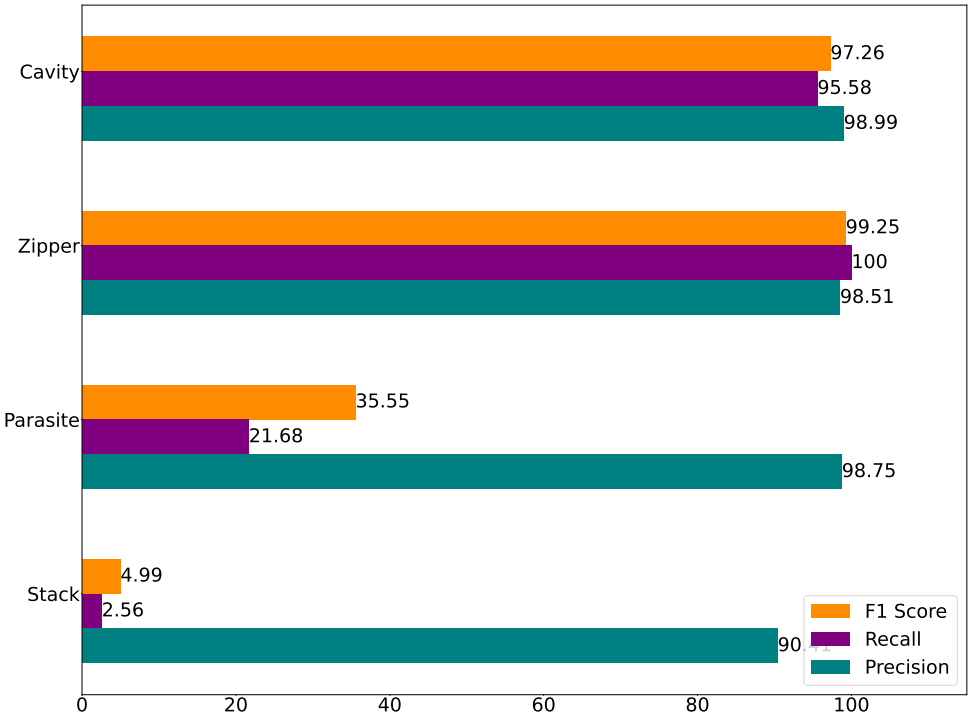


Figure 2.2: `file` performs well on the rare zippers and cavities, but recall is extremely low on the more common stacks and parasites.

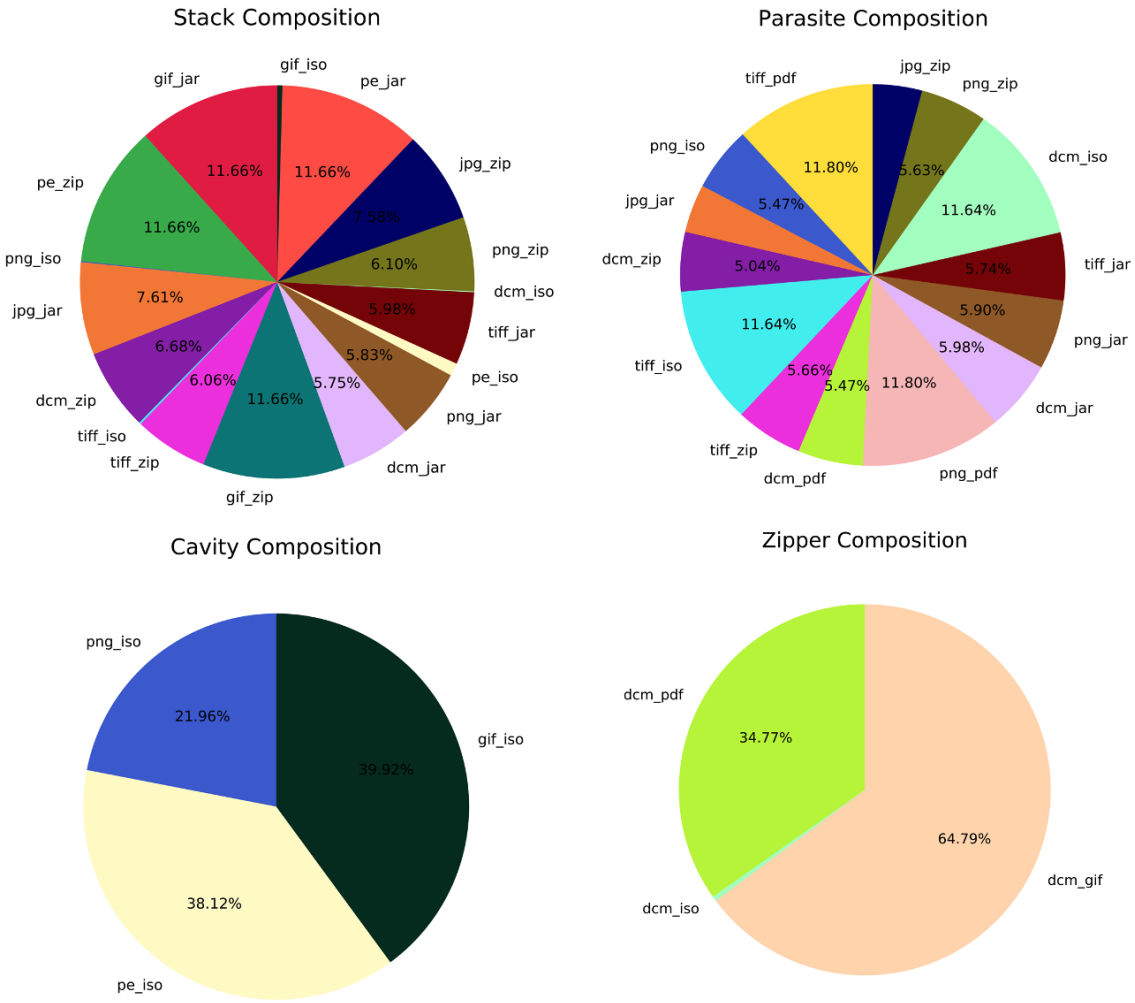


Figure 2.3: File combinations that could be used to create each polyglot type from the monoglot dataset. Note that Cavities and Zippers had many less viable file combinations using our donor file types, which we hypothesize made them easier for file to detect.

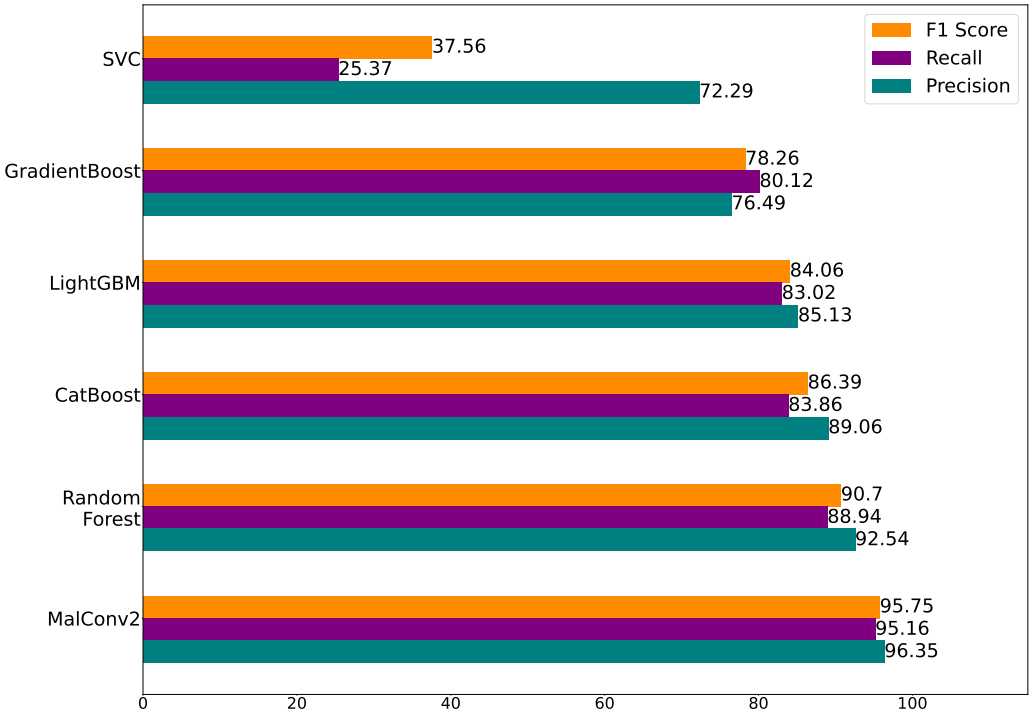


Figure 2.4: MalConv2, the deep learning model trained on raw bytes, has the best performance in terms of recall, precision, and F1 Score. The remaining models are trained on byte histograms extracted from the files as described in Section 2.2.3.2.

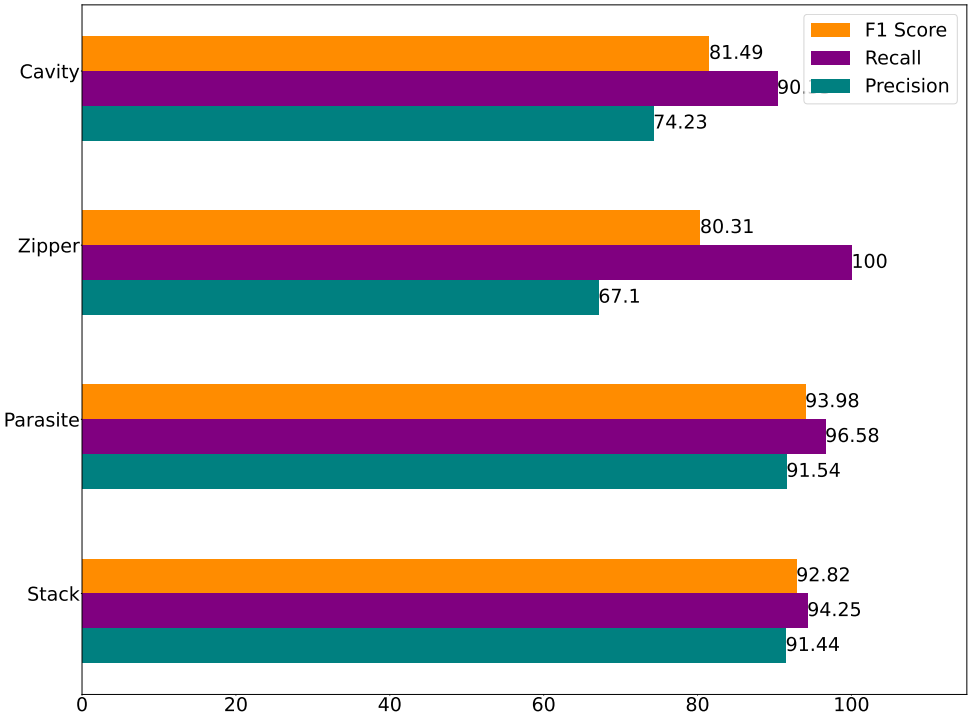


Figure 2.5: Detailed look at MalConv2's performance by polyglot combination method. Recall is high across the board, although precision is lower for cavities and zippers.

Next, we added the mime-type output generated by `file` as an additional feature in our ML models. Figure 2.6 shows the precision, recall, and F1 score of the ML models when the feature vector is a concatenation of the byte histogram and the mime-type for each file. The mime-type is converted to a 1-hot encoding for ingestion by the model. This additional feature boosted the recall of our machine learning models across the board. We tested whether adding this feature to MalConv2 would likewise boost the deep learning model's performance; however, the additional feature merely degraded MalConv2's performance. The performance of CatBoost improved dramatically from 83.86% to 94.75%. Due to CatBoost's improvement, we tuned this model's hyper-parameters to see if we could improve performance even more, yielding our best recall to date: 95.45.

Figure 2.7 provides a close-up look at the tuned CatBoost model's performance by polyglot combination method. The results are quite similar to MalConv2's in that recall is consistently high but precision is lower on zippers and cavities.

Figure 2.8 depicts the receiver operating characteristic (ROC) curve for the top-scoring models. The area under curve (AUC) for each model is displayed in Table 2.2. As with the other metrics, the tuned version of CatBoost has the best score, followed closely by MalConv2.

The hyper-parameter tuned version of CatBoost narrowly outscored MalConv2 (95.45 vs 95.16 recall) thanks to the additional mime-type feature. We tried adding the mime-type to MalConv2, but it did not result in a meaningful improvement. The CatBoost model, like all of the ML models, has the advantage of a compact feature space compared to MalConv2, which expects raw bytes. Our version of MalConv2 truncates an input file if the file exceeds 16MB in size. Therefore, the feature vector is quite large (16MB) and training is much slower than when using the compact feature vector with CatBoost. Although much work remains to be done in terms of optimizing the detection of polyglot files, we currently recommend the use of a CatBoost model whose feature space consists of the byte histogram concatenated with the mime-type

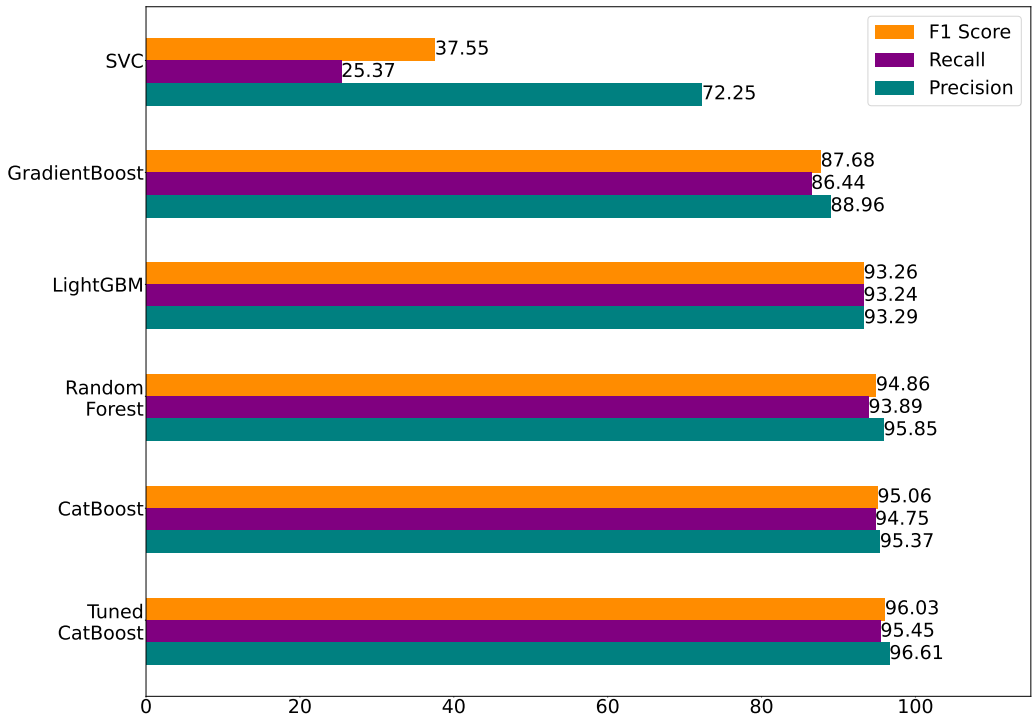


Figure 2.6: When the feature vector of the ML models was enhanced via the addition of the `file` utility's mime-type output, performance increased across the board. Since CatBoost improved significantly, we conducted hyperparameter tuning to produce a version which narrowly outperforms MalConv2. We tested whether adding the mime-type feature to MalConv2's raw byte feature vector would improve that model as well. It did not. Ergo CatBoost is now the top-scoring model and has the added benefit of using a much smaller feature vector than MalConv2.

Table 2.2: AUC Scores for Top Models

Model	AUC
Tuned CatBoost	.994
MalConv2	.988
Random Forest	.969
CatBoost	.957
LightGBM	.938
GradientBoost	.891

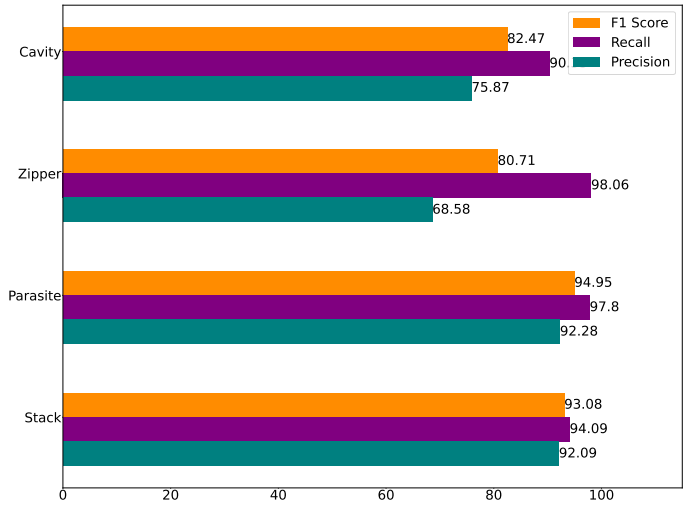


Figure 2.7: Detailed results for tuned Catboost model. Recall is consistently high, but precision is lower on zippers and cavities.

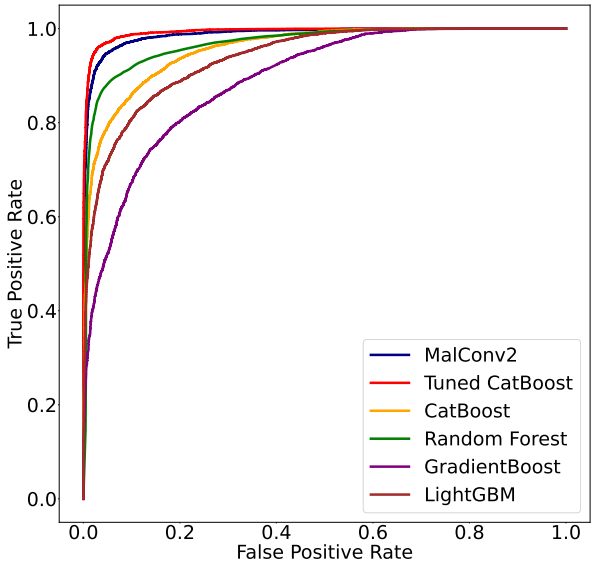


Figure 2.8: ROC curve for top scoring models. The tuned version of CatBoost achieves the best performance followed closely by MalConv2.

output of `file` (converted to a 1-hot encoding) for polyglot detection. This model had the highest recall combined with an efficient feature space.

2.2.4.3 Comparing ML and DL

While we used the raw bytes of each file as the input for Malconv deep learning model, the feature space for our ML models consisted of the byte count histogram and mime type of each file. One could argue that this is an apples-to-oranges comparison. However, our objective was to compare disparate approaches to solving the same problem rather than how two different machine learning algorithms handle a set of input features. Decision trees tend to overfit when the feature space is large, so using the byte histograms (or some other method of reducing the feature space) is the logical choice. In contrast, deep learning is often an effective approach when the feature space is large and is capable of creating its own intermediate representations of the data, so using the raw bytes is a logical choice for deep learning. In addition, Malconv2 is specifically designed to work with raw bytes.

2.2.4.4 Detection on Novel Polyglots

The goal of training these models is to be able to detect novel polyglots created by malicious actors with the same techniques and file types that we used to create our dataset. We were not attempting to detect entirely novel polyglot creation techniques, which would require an unsupervised or semi-supervised approach geared more towards anomaly detection. Did we achieve this goal?

We did test both the our models on a small dataset of malicious polyglots created by a professional red team (we cannot release details due to an NDA), with the CatBoost model achieving the best results with 99.4% accuracy. MalConv2 only achieved 83.80% accuracy on this validation set, indicating that CatBoost generalized better on held-out data. Malconv2 likely experienced overfitting; further testing on a larger dataset is part of our future work. In general, decision tree based methods

that used boosting outperformed those that used bagging, with random forest only achieving 75% accuracy. These initial results suggest that our models can generalize, but additional testing is needed on larger datasets to verify performance across file types and different categories of polyglots.

2.2.5 Conclusion and Future Work

We demonstrated that commonly used utilities fail to reliably detect polyglots. We then evaluated machine and deep learning models for detecting polyglots and found that MalConv2 initially had the highest recall. If mime-type is included as a feature, then a hyper-parameter tuned version of CatBoost has the highest recall. CatBoost generalized to a validation data set better than MalConv2 and has the added advantage of a much smaller feature vector. Our work thus far is merely the beginning, however. There are several avenues of research that need to be advanced in order to produce an operational polyglot detector, namely:

- Data diversification: A production-grade system can encounter an extremely wide variety of file formats; ergo, We need to assemble a data set that fully encompasses the wide variety of file formats commonly encountered in the wild.
- Feature space optimization: We need to asses and possibly incorporate additional features into our feature vector. Historical candidates include byte co-occurrence counts and byte entropy.
- Hyper-parameter tuning: We need to continue to optimize the top-scoring models to achieve maximum performance.
- High throughput: We need to demonstrate that our top-scoring models have high enough throughput to be incorporated into production-grade systems where high throughput is essential.

- Benign polyglots: We need to address instances where files are combined into polyglots for benign purposes, e.g., DICOM+TIFF files that combine medical images into one file.

2.3 On the Abuse and Detection of Polyglot Files

2.3.1 Introduction

A polyglot file simultaneously conforms to two or more file-format specifications. This means the polyglot file can exhibit two completely different sets of behavior depending on the calling program, as depicted in Figure 2.9. This dual nature poses a threat to endpoint detection and response tools (EDR) and file-upload systems that rely on format identification prior to analysis. As shown in Figure 2.10, a polyglot can evade correct classification by first evading format identification. If only one format is detected, then the sample may not be routed to the correct feature-extraction routine (in the case of machine learning-based detectors) or compared to the correct subset of malware signatures (in the case of signature-based malware detection). As evidence that existing commercial off-the-shelf (COTS) endpoint detection and response tools are vulnerable to polyglots, we point to Bridges et al. Bridges et al. (2020), who demonstrated that 4 competitive COTS tools detected 0% of the malicious polyglots in the test data.

Standardized formats for files play a key role in cybersecurity. By first identifying the format of an unknown sample, they allow malware detection tools to extract the most discriminate and robust features from an unknown sample. This allows the detection tool to discard unimportant bytes that can be manipulated to alter classification in an adversarial attack Kolosnjaji et al. (2018a); Demetrio et al. (2021b). However, this feature-extraction process introduces a vulnerability; the correct format must be detected in order to route the file to the correct feature extractor. Even when a detector does not use machine learning and instead relies upon signatures

for detection, the need to maintain a high throughput encourages EDR tools to only search for signatures that correspond to the detected format Jana and Shmatikov (2012a).

As prior researchers Jana and Shmatikov (2012a); Albertini (2015); Clunie (2019); Ortiz (2019); Desjardins et al. (2020); Popescu (2012c) have demonstrated, polyglot files can be crafted that are fully valid (execute as intended) in multiple formats. To date, however, no comprehensive study of polyglot usage by malicious actors in the wild and/or methods of detecting said polyglots has been undertaken. In this paper, we set out to answer four key research questions related to polyglot usage and mitigation:

RQ1: *How are polyglots currently used by threat actors in the wild? This includes the role the polyglot fills, the formats of the donor files, and the combination method used to fuse the donors together.*

RQ2: *Can we train a detector to effectively filter or reroute polyglots prior to ingestion by a malware detection system?*

RQ3: *Does this detector outperform existing file type detection, file carving, and polyglot-aware analysis tools at detecting polyglot files?*

RQ4: *Given the prevalence of image-based polyglots in adversary usage and the relative simplicity of image formats, what tools can we provide to defenders to address image-based polyglots in their existing workflows?*

To address **RQ1**, we reviewed open-source intelligence feeds (see Section 2.3.3.1 for methods) that detail adversary tactics, techniques and procedures (TTP), finding that polyglots have played an important role in a number of malicious campaigns by well-known advanced persistent threat (APT) groups. Polyglot files allowed the malicious actors to covertly execute malicious activity and extract sensitive data by masquerading as innocuous formats. In Section 2.3.3, we provide an overview of the different roles polyglots played in each campaign, detail the file combinations used, and provide a detailed description of several high profile examples. To address **RQ2-RQ4**, we first created a tool, *Fazah*, for generating polyglots that mimic the examples seen

in the wild. Although there are other possible format combinations, our goal with this tool was to mimic, as closely as possible, the formats and combination methods used by real-world threat actors. Using this tool, we then created a data set of polyglot and normal (referred to hereafter as monoglot) files for training and testing. See Section 2.3.4 for a full description of the data set.

To address **RQ2**, we tested machine learning models to solve both the binary and the multi-label classification problems, achieving an F1 score of 99.20% for binary classification and 99.47% for multi-label classification with our deep learning model PolyConv. To address **RQ3**, we evaluated five commonly used format identification tools on this dataset: *file* Darwin et al. (2019), *binwalk* ReFirmLabs (2021), *TrID*, *polydet*, and *polyfile*. These tools were selected because of their use in existing cybersecurity tools or claim to detect polyglot files. We evaluated the performance of these tools at both binary and multi-label classification. In our context, binary classification determines whether a file is a polyglot or a monoglot. Multi-label classification, on the other hand, identifies all formats to which the file conforms. We found that existing tools did not exceed an F1 score of 93.32% at binary classification and 83.74% at multi-label classification.

See Section 2.3.5 for details regarding our ML based approaches and Section 2.3.6 for a comparison of ML-based approaches to existing file-format identification tools.

As detailed in Section 2.3.7, to address **RQ4** we developed and tested a CDR tool for sanitizing image-based polyglots since these were the most common vector for polyglot malware. We also tested YARA rules for detecting extraneous content in image files. We found that the YARA rule approach did not generalize well to all formats that can be combined with an image, especially the more flexible scripting formats like Powershell or JavaScript. However, they may be use in high-throughput use cases where deploying a deep learning model is not feasible. A more effective approach is to strip all extraneous content from images using a content disarmament and reconstruction (CDR) tool. Our CDR tool, *ImSan*, was able to sanitize all of the

image polyglots in a random subset of our image polyglots. A subset was used so we could manually verify the results.

The following provides a summary of our contributions:

- **RQ1:** The first, to our knowledge, survey of polyglot usage by malicious actors in the wild, demonstrating that polyglot files are an actively used TTP by well-known malicious actors. Utilizing the results of this study, we created a tool, *Fazah*, to generate polyglots using formats and combination methods exploited by malware authors in the wild. We then used *Fazah* to generate a dataset of polyglots and monoglots to evaluate existing detection methods and train polyglot detection models.

- **RQ2:** Utilizing this novel dataset, we trained a deep learning model, PolyConv, that can distinguish between polyglots and monoglots with an AUC score over 0.999. We also created a multi-label model that reports all of the detected formats in monoglot and polyglot files, enabling analysts to quickly determine the nature of a threat or route the suspicious file to multiple format-specific detection systems.

- **RQ3:** We provide a comparison of our polyglot detection models with existing file-format identification and carving tools, some of which are polyglot aware. This evaluation shows that existing methods for detecting file type manipulation are inadequate and often fail to detect polyglot files, even with special flags set that are meant to ensure multiple file types are detected.

- **RQ4:** For image-based polyglots, which are common in the wild, we explored YARA rules and content disarmament and reconstruct (CDR) tools, finding that our *ImSan* CDR tool was 100% effective while the YARA rules did not compete with our deep learning detector. They may, however, be of use in high throughput situations.

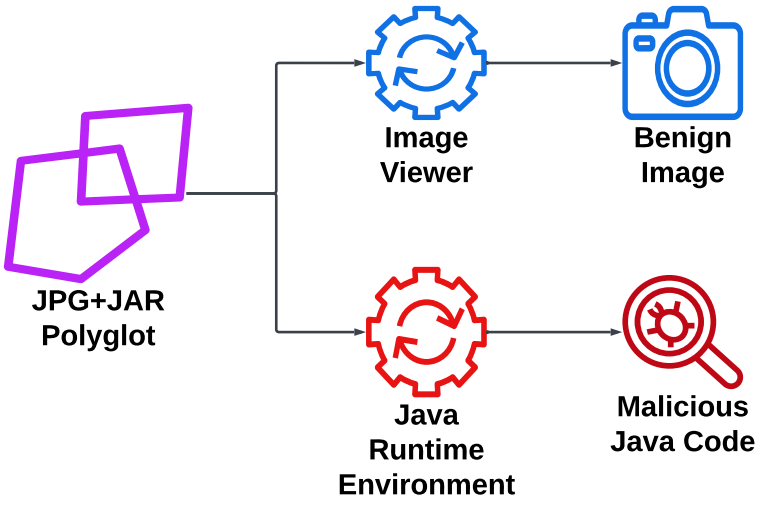


Figure 2.9: Functionality of a polyglot file is determined by the calling program, which can be explicitly provided or automatically determined by the operating system's auto-launch settings.

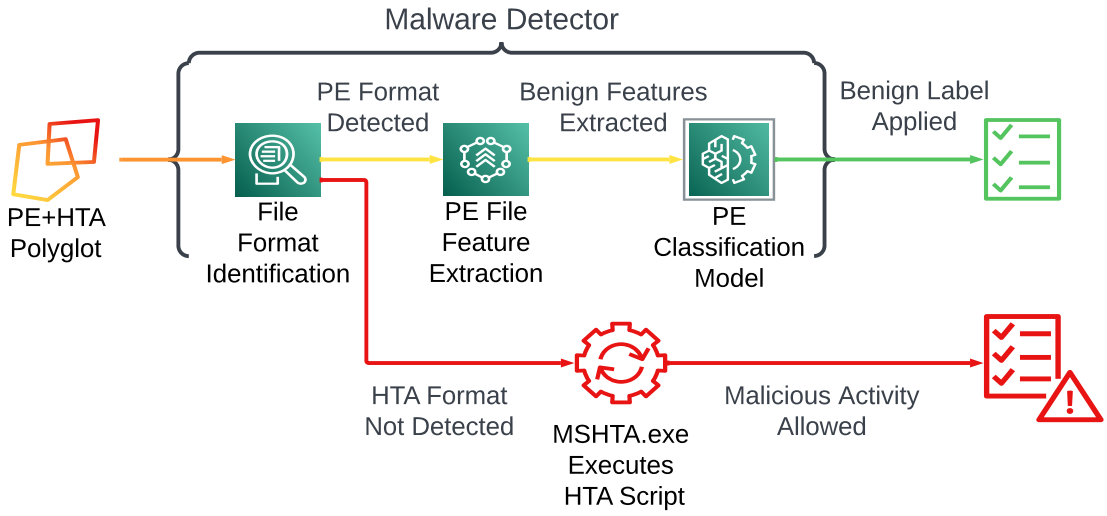


Figure 2.10: Since polyglot files simultaneously conform to multiple formats, they can evade correct format identification. This in turn allows them to evade format-specific feature extraction or signature matching, thereby evading malware detection. Therefore, some preprocessing should be done to either filter/quarantine polyglot files prior to feature extraction or route them to multiple format-specific malware detectors so all functional components of the polyglot are analyzed.

2.3.2 Related Work

2.3.2.1 Polyglot Detection

Bridges et al. conducted an in-depth evaluation of four leading COTS tools Bridges et al. (2020). Among the test data were 199 malicious JPG+JAR polyglots that went completely undetected by all 4 tools. While we can not prove why these tools failed across the board, we can surmise—based on the unusual 0% detection rate—that the failure occurred in the file-type identification that must occur prior to feature extraction. If the files were interpreted as JPG (the benign component) rather than JAR (the malicious component), it is unlikely that the malicious JAR content was analyzed. This provides a plausible explanation for the complete detection failure. Therefore, to solve the problem of malicious polyglot detection, the problem of correct file type identification should first be solved.

The machine learning models FiFTy and Sceadan—a support vector machine (SVM) and a deep learning model, respectively—were released by researchers for file-format identification Mittal et al. (2020); Beebe et al. (2013b). However, neither tool was designed with polyglots in mind nor trained on a dataset containing them.

2.3.2.2 Polyglot Creation

Jana and Shmatikov demonstrated a number of attacks that exploited discrepancies in file type inference and file parsing. Specifically, they found that polyglots—referred to as "ambiguous files conforming to multiple formats" Jana and Shmatikov (2012a)—evaded detection by 20 out of 36 malware detectors. Using the open-source ClamAV tool as an example, they point out that malware detectors may terminate format inference at the first match, extracting features and/or checking malware signatures only for the first format detected in the polyglot. Jana and Shmatikov argue that exhaustively testing incoming files against all possible formats would introduce an unacceptable overhead. Flexibility in format specification and parser tolerance of

malformed files are also presented as reasons why simply improving existing tools is difficult.

Ange Albertini demonstrated that a wide variety of files can be combined into polyglots Albertini (2015). He created an open-source tool known as *mitra* that can create 4 types of polyglot from a wide range of files. He defined the four types thusly:

- *Stack*: File B is appended to the end of file A
- *Parasite*: File B is placed inside comment markers of file A
- *Zipper*: Both files are placed within one another's comment markers
- *Cavity*: File B is placed inside a padding area of file A

2.3.2.3 Polyglot Exploitation

A number of previous academic works demonstrated the risk polyglots may pose. For example, a DICOM file is an image archive format designed for medical use. The format was designed to be flexible so medical staff could combine a variety of image formats into a single file for a patient Graham et al. (2005). However, that flexibility means a DICOM file will tolerate combination with a Windows Portable Executable (PE) file to create a malicious polyglot Ortiz (2019). This polyglot could allow an adversary to propagate their malicious PE through a medical network, activating the PE component through a second stage of the attack.

In an attack on data integrity, Popescu demonstrated that a PDF+TIFF polyglot can bypass digital certification verification Popescu (2012a). In this scenario, the attacker sends a valid request (PDF file) for a bank transfer to a target. The attacker's goal is to change the amount authorized in the PDF without invalidating the certification applied by the target. When the victim opens the file, auto-launch settings intepret the file as a PDF and present the legitimate PDF contents to the victim.

The victim then applies a digital signature that protects the file contents from any future change, and returns the file to the attacker. However, the file is also a TIFF file. The TIFF is an image of the same PDF, albeit with a much larger money transfer authorized. The attacker does not edit the contents of the file (which would break the signature). They merely change the file extension, switching the auto-launch behavior from opening the PDF contents to opening the TIFF contents, before sending the file on to a hypothetical bank.

Since the file contents have not changed, the digital certification is still valid. When the bank opens the file, auto-launch behavior shows the larger fraudulent TIFF transaction rather than the proper PDF amount the victim agreed to when they signed the file.

2.3.3 RQ1: Polyglot Exploitation in the Wild

Thanks to Bridges et al. Bridges et al. (2020), we know polyglots can evade detection by COTS tools. However, the extent to which malicious actors employ polyglots has never, to our knowledge, been published before. Do malicious actors use polyglots in their attack chains? What role do polyglots play within an attack chain? What file formats and combination methods were utilized in these attacks? To address these questions we conducted a survey of threat intelligence feeds, collecting file hashes of polyglot samples and information on the roles played by these files within attack chains. For the file hashes and a list of the sources used in this survey, see Tables 6.1, 6.2, and 6.3 in Section 6.2.

2.3.3.1 Survey Methods

The survey, performed between November 2022 and January 2023, focused on identifying the role of a polyglot file within a threat actor's cyber-attack chain. We used publicly available independent sources, general search engines and threat intelligence feeds (e.g., ORKL, X) to gather a wide range of information security

reports and articles. Those sources were searched using the following terms: *polyglot*, *combined*, and *contained*. We found that the term *polyglot* is not always utilized in reports. We therefore had to manually distinguish between reports of true polyglots (two or more valid formats in one file) and other forms of digital steganography. A number of reports described malware that contained a valid format along with an oft-encrypted set of malicious instructions. We do not consider these files as polyglots because the malicious instruction can only be correctly interpreted when passed as input to another component of the malware rather than a parser conforming to a published standard.

For each true polyglot found, we used our knowledge of threat operations to determine the role the polyglot played in the cyber-attack chain. Lastly, the online malware databases, VirusTotal and MalwareBazaar, were used to obtain the actual polyglot samples whenever hashes of the polyglot were provided in a report. The file hashes and sources from our survey of open-source intelligence can be found in the appendix in Tables 6.1, 6.2 and 6.3, respectively.

2.3.3.2 Role of Polyglot Files in Cyber Attack Chains

The survey discovered fifteen examples of a threat actor using a polyglot file in their cyber-attack chain, along with 30 distinct polyglot files. According to MITRE's Adversarial Tactics, Techniques, and Common Knowledge (ATT&CK) framework, polyglots are primarily utilized for *Defense Evasion* (MITRE ATT&CK TA0005). Polyglot files also fall under the *Obfuscated Files or Information* (MITRE ATT&CK T1027) heading since these files conceal hidden functionality by appearing to conform to only one file format. We obtained 30 polyglot samples from VirusTotal and MalwareBazaar using the file hashes specified in the reports.

For the purpose of establishing a formal taxonomy for polyglot files, we refer to polyglots as having an overt format and a covert format. The overt format is the format the file presents as (e.g., matches the extension) while the covert format is not apparent without analysis. In most cases, a polyglot consists of a malicious file combined with

a benign one; however, in some cases we found that both file formats play a role in advancing the malicious attack chain, as in the HTA+CHM polyglot utilized by IcedID in Section 2.3.3.2.1. Therefore, we instead refer to polyglots as combining an overt format with a covert format. A summary of the found-in-the-wild samples is provided in Table 2.3. In Appendix 6.2 we discuss the capabilities of interest that each file format provides to the malware author (camouflage, non-standard execution path, etc.) to understand why these combinations exist in the wild and how they fill a desired role in attack chains.

We selected two cyber attack chains to demonstrate how well-known APTs utilize polyglots to reach the next step in their cyber attack chains. A third attack chain can be found in Appendix 6.2. CVE numbers and MITRE ATT&CK references are provided where applicable.

2.3.3.2.1 IcedID

IcedID is a banking trojan that, according to Check Point's Global Threat Index, was the fourth most widespread malware variant in 2022 Team (2022a). The trojan uses an evolving variety of methods to establish initial access. One of these methods relies on a polyglot formed by combining a CHM and an HTA file.

The attack chain is illustrated in Figure 2.11. It begins with a password-protected Zip file attached to a phishing email. The Zip contains an ISO file which exploits CVE-2022-41091 to evade flagging by Microsoft's *alternate data stream* (ADS) defensive mechanism Hegt (2020).

The ISO file in turn contains two files: a DLL (hidden by default on Windows) and a CHM+HTA polyglot. The polyglot masquerades as a CHM file which presents a benign decoy window when executed. The Microsoft compiled HTML (CHM) format used for software documentation. Each file consists of a number of HTML pages organized into a document that is compressed into a binary stream. As with any HTML page, CHM files may download/execute other files or run Powershell/Javascript commands when viewed.

In the background, this CHM file starts a MSHTA.exe process with itself as the input. This new process executes the malicious component of the polyglot, the HTA file, which in turn launches the hidden DLL file that contains the actual IcedID payload.

2.3.3.2.2 Andariel/Lazarus

Lazarus (of which Andariel is a subgroup) is an advanced threat group that has operated out of North Korean since 2009 Htet (2017). In 2021 attack chains connected to this group utilized polyglots to infect systems with a Remote Access Trojan (RAT) Jazi (2021); Park (2021); this process is illustrated in Figure 2.12.

This attack chain typically begins with a phishing email that has an attached malicious Microsoft Word Document (DOC) file (MITRE ATT&CK T1566). When the DOC file is launched, a macro begins execution (MITRE ATT&CK T1204.002). First, the macro drops a PNG file to the *Temp* directory. The image data in the PNG file is a compressed polyglot file.

Next, the DOC macro converts the PNG file to a BMP file, which has the intended side effect of decompressing the contents (MITRE ATT&CK T1140). The DOC Macro does this by leveraging the Windows Image Acquisition (WIA) Automation Layer Objects: ImageFile and ImageProcess Microsoft (2018a,b).

After conversion, the DOC Macro saves the BMP as a zip file by giving it a zip extension. However, the file is actually a BMP+HTA polyglot, with the HTA covert contents appended to the end of the overt BMP data. Finally, the DOC Macro executes the polyglot file as an HTA file using the MSHTA application via the Windows Management Instrumentation (WMI) Service (MITRE ATT&CK T1059, T1047).

WMI is used so that the resulting process does not appear to be a child of the DOC process. The HTA file drops its payload, a hidden PE file, into a hidden folder. Finally, the HTA file launches the PE file which provides a foothold on the target system for future exploitation.

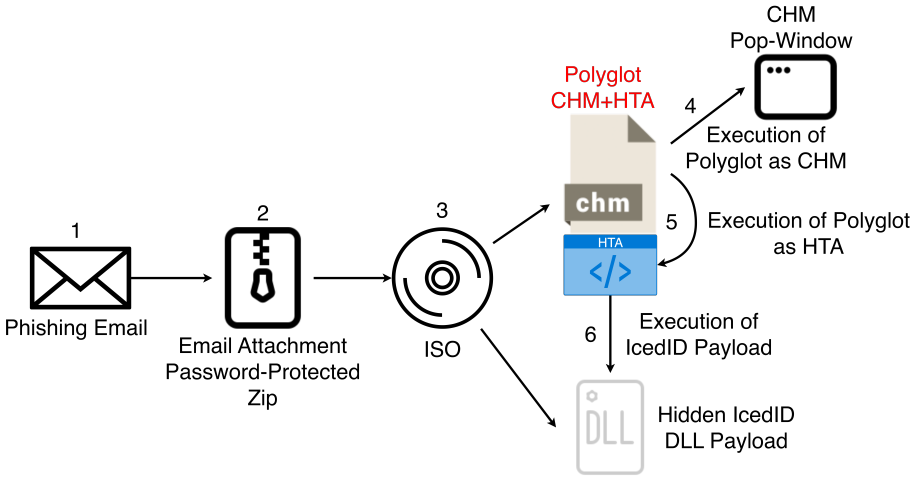


Figure 2.11: IcedID Attack Chain

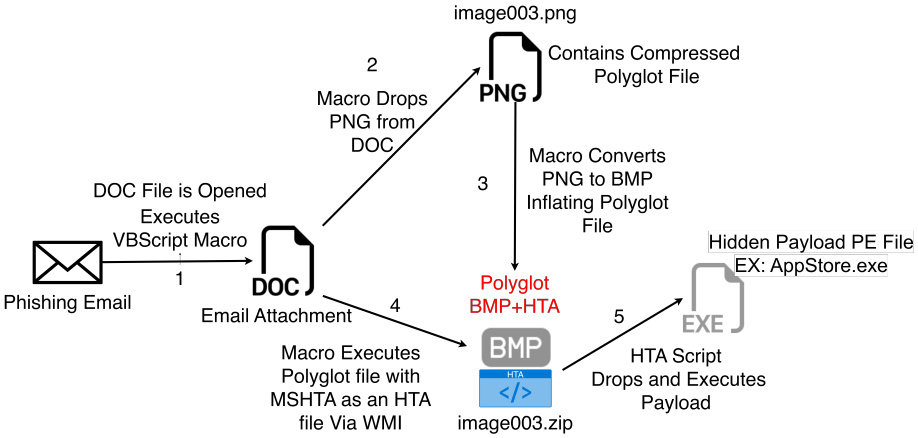


Figure 2.12: Andariel/Lazarus Attack Chain

2.3.4 Wild Polyglots: A Polyglot Data Set Based on Malicious Usage in the Wild

This section describes how we created our data set based on our survey of polyglot usage in the wild (**RQ1**) using the *Fazah* tool in order to address **RQ2-RQ4**.

2.3.4.1 Fazah: A Polyglot Generation Framework

Having uncovered which formats have been used in real-world malicious polyglots, we created a data set consisting of monoglot and polyglot files conforming to these formats. Our first step was to create a framework for generating polyglots by combining donor files. Our goal for this tool was to mimic format and combination methods found in the wild rather than demonstrate all possible combinations. The *Fazah* framework is a modular tool written in Python that can currently generate 46 format combinations using 8 *covert* formats. The combination method—*stack* and a variety of *parasites*—is derived from reports of malicious use in our survey and varies between *covert* format. As discussed in the survey, malicious actors use polyglots either to disguise malicious content using a less suspicious format (images) or add hidden functionality (scripts). Since image formats typically use comment markers, *parasites* are commonly used by malicious actors. *Stacks*, meanwhile, are the simplest and easiest method for malicious actors to implement, working well with script and archive formats. Files with distinct comment markers (necessary for *zippers*) are quite rare. Of the common (but by no means exhaustive) set of formats we tested, only DCM combined with either PDF/GIF/ISO could result in a *zipper*. Similarly, we found that only ISO paired with PE/PNG/GIF yielded *cavities*. This does not preclude their use in malicious campaigns, but places them beyond scope for our goal of emulating known attack chains. Table 2.3 provides the format pairings that *Fazah* can turn into polyglots. Given the possibility for malicious abuse of the framework, *Fazah* will not be published publicly at this time.

2.3.4.1.1 Wild Polyglots Data Set Creation and Contents

We collected benign files conforming to 13 common formats using Github's search API: BMP, EXE, GIF, HTA, JAR, JPG, JS, MSI, PHP, PNG, PS1, RAR, ZIP. Using a held-out set of donor files, we created 32 types of polyglots organized according to which 2 types of donor files were combined to create the polyglot file. We kept all donor files separate from the train and test set to ensure that the models did not cheat by learning that data added to a monoglot in the training set is a polyglot. Table 2.4 provides an overview of the Wild Polyglots data set. Figures 2.13 and 2.14 breakdown the formats contained in the monoglot and polyglot training sets, respectively. Since our objective was to train a polyglot detector rather than a malware detector, we only utilized benign files. We first scanned the files we scraped for malware and removed any suspicious samples. Next, we removed any scraped files whose extension did not match the file contents (e.g., a JPEG with a .png extension) or if the file could not be parsed by an appropriate utility (e.g., Pillow for images). We erred on the side of inclusion for highly flexible scripting language formats like HTA. Since MSHTA.exe is tolerant of a high degree of malformation, we felt it unwise to exclude malformed HTA from our training data.

2.3.5 RQ2: Using Machine Learning for Polyglot Detection

This section explores using machine learning to detect polyglot files. Section 2.3.5.1 chronicles our development process as we tested different ML model architectures and experimented with improvements to the feature space. Section 2.3.5.2 presents the results from out best-performing models compared to existing tools.

2.3.5.1 Ml-based Detection Development

Our first objective was to determine which machine learning architecture and feature set were most effective at detecting polyglots. Toward this end, we created a small ($\sim 70,000$ files) initial data set using the *mitra* tool (described in Section 2.3.2.2)

Table 2.3: Polyglot Formats Deployed Maliciously in the Wild

Covert Format	Overt Format
HTA	JPEG, PNG, BMP, GIF, LNK, PE, MSI, RAR, Zip, TTF, RAR, CHM, PDF
PHP	JPEG, PNG, BMP, GIF, TTF, RAR, Zip, LNK, PDF
PHAR	JPEG, PNG, BMP, GIF
JavaScript	GIF, BMP
PowerShell	JPEG, BMP, GIF
Zip	JPEG, PNG, GIF, PDF
JAR	JPEG, PNG, GIF, PDF, MSI
RAR	JPEG, PNG, BMP, GIF
BMP	Zip, JAR

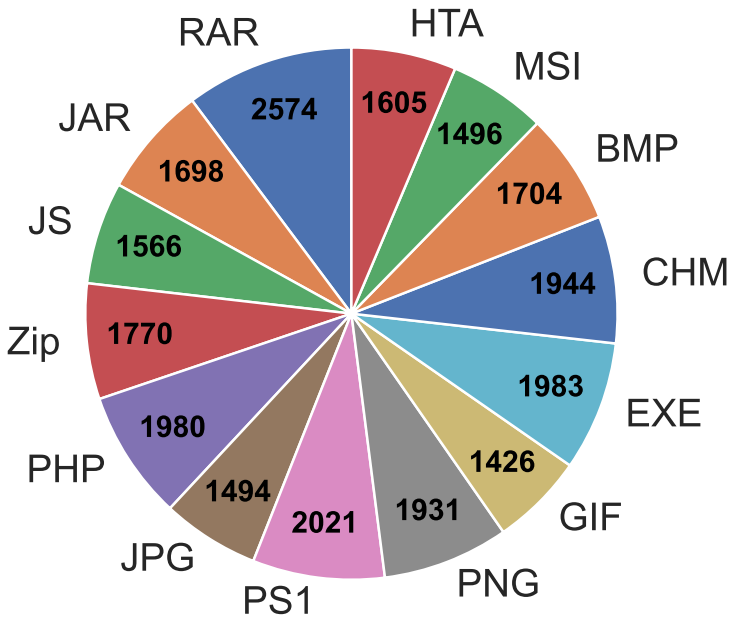


Figure 2.13: File counts for the monoglot formats in the Wild Polyglots training data.

Table 2.4: Wild Polyglots Data Set Contents

	Train	Test
Monoglot	25192	9975
Polyglot	1148604	213109

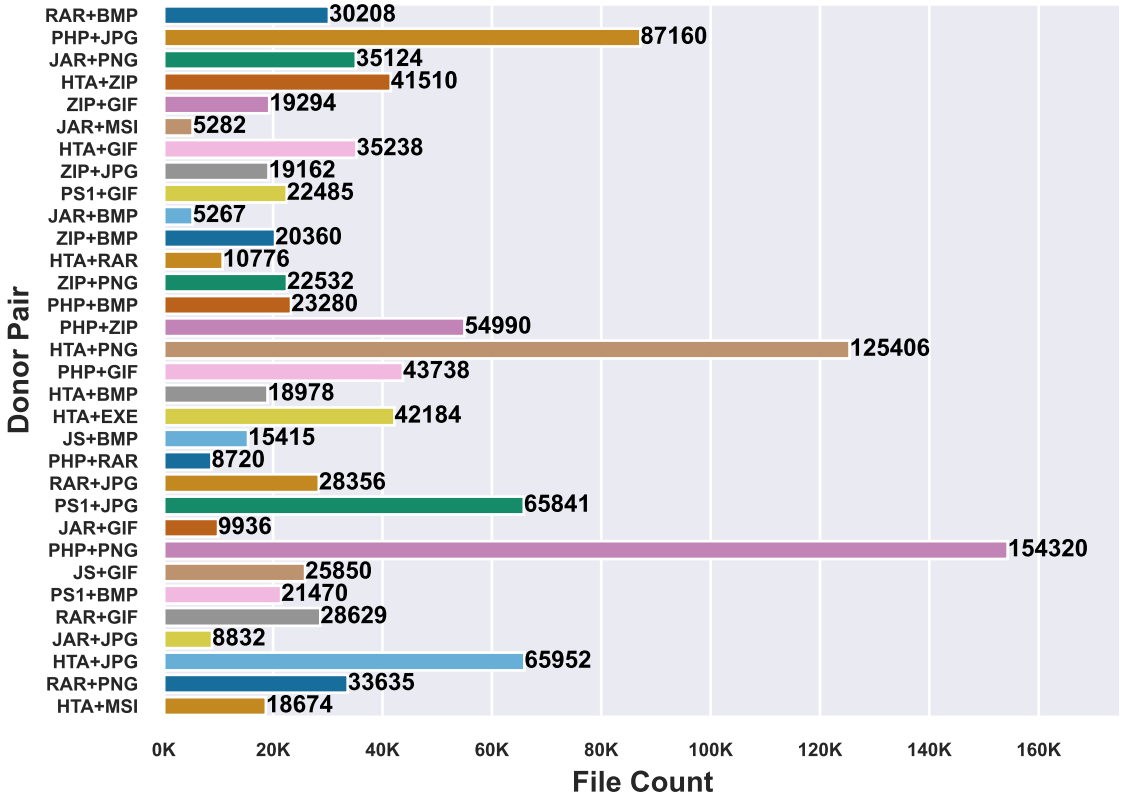


Figure 2.14: File counts for each of the 32 polyglot combinations in the Wild Polyglots training data.

prior to the development of our *Fazah* tool. On this preliminary data set, we tested a Support Vector Machine, Random Forest, GradBoost, CatBoost, LightGBM, and MalConv. With the exception of MalConv Raff et al. (2018a), these models used the byte histogram as their only feature. The byte histogram is a vector of length 256 where the value stored at each index corresponds to the number of times that byte value occurs in the input file. This feature vector is agnostic with respect to file formats since all digitally stored files are a string of bytes. We found that, on this preliminary data set, MalConv and CatBoost were the top performers.

We focused further development on MalConv and CatBoost, labeling our improved versions PolyConv and PolyCat, respectively. At this point, we trained and tested both models on our survey-informed Wild Polyglots data set; all results and figures reported in this paper refer to the Wild Polyglots data set. We found that, for PolyCat but not PolyConv, adding the mime-type output of the *file* utility improved results. Although *file* was not competitive at detecting polyglots (see Section 2.3.6.1) or at identifying both formats contained within, it was extremely accurate at identifying the first format contained in the file. Therefore, we augmented PolyCat's feature space with a 1-hot encoding of the mime-type output from *file*. We found further improvement by adding the 8000 most common bigrams and trigrams extracted from each file using an overlapping window. Thus, the final feature space for PolyCat consisted of the byte histogram, the 1-hot encoding of the mime-type from *file*, and the most common bigrams and trigrams.

MalConv is an oft-cited deep learning classifier designed to detect malware Raff et al. (2018a). We trained the model from scratch to identify polyglots rather than to identify malware. None of the polyglots in our data set were malicious in order to guarantee that the model learned to detect multiple formats rather than malicious content. Since the model is trained on raw bytes rather than format-specific features (e.g., the EMBER feature set for PE files Anderson and Roth (2018a)), MalConv's architecture is well-suited to the polyglot detection problem which requires a format-agnostic approach. In lieu of a fixed feature-extraction routine, the model takes in raw

bytes and learns an encoding (first layer) as well as a set of filters (the convolution layers) to recognize significant byte patterns. MalConv also features an attention and gating mechanism intended to filter out extraneous information in the raw bytes.

We experimented with changes to the architecture in order to make it more effective at our novel task, yielding the PolyConv model mentioned above. The original architecture of MalConv is presented in Figure 2.15 while PolyConv's architecture is presented in Figure 2.16.

The changes we made to MalConv consist of the following:

- Decreasing the window and stride from 512 bytes to 16 and 8 bytes, respectively, in order to capture the byte patterns of very short (in terms of bytes) script files hidden within larger files
- Removing the attention and gating mechanism as they did not seem to improve the results on our task
- Increasing the number of kernels in the remaining convolution layer to 512 in order to learn enough byte patterns to distinguish the wide variety of distinct formats upon which we trained the model
- Increasing the number of fully connected layers to 3 as a result of experimenting with different layers counts
- Increasing the number of nodes in each fully connected layer to 512, 512, and 128 as a result of experimenting with different node configurations

2.3.5.2 Comparing ML-based Polyglot Detection Approaches

We trained and tested PolyConv, MalConv, PolyCat, and CatBoost on our Wild Polyglots data set. For this comparison, we evaluated binary label (polyglot or monoglot) versions of the models. Since our data set is imbalanced, we used the precision-recall curve rather than the ROC curve to score our models. Therefore, our top model is the one with the highest PR-AUC on the Wild Polyglots test

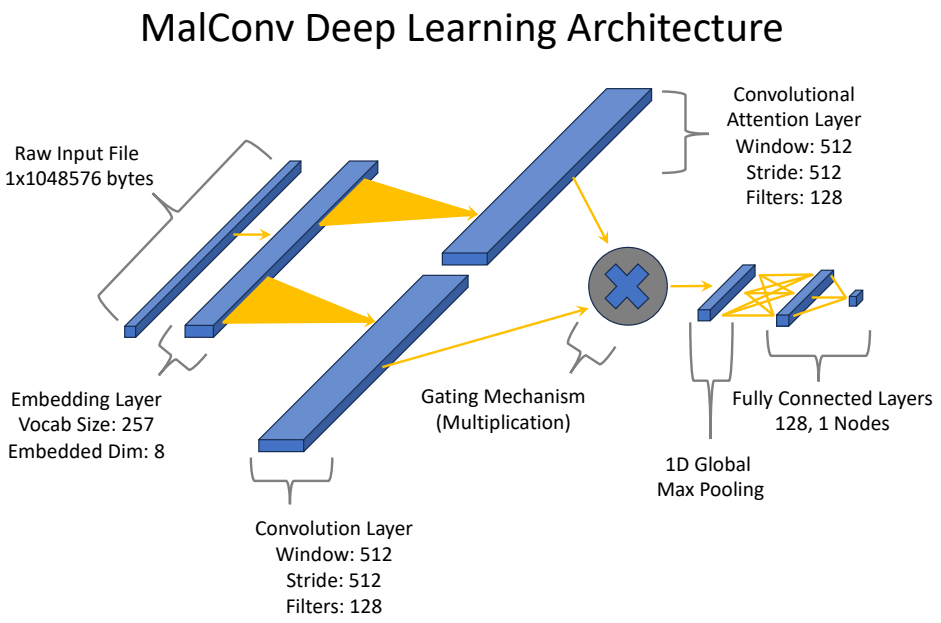


Figure 2.15: MalConv Architecture

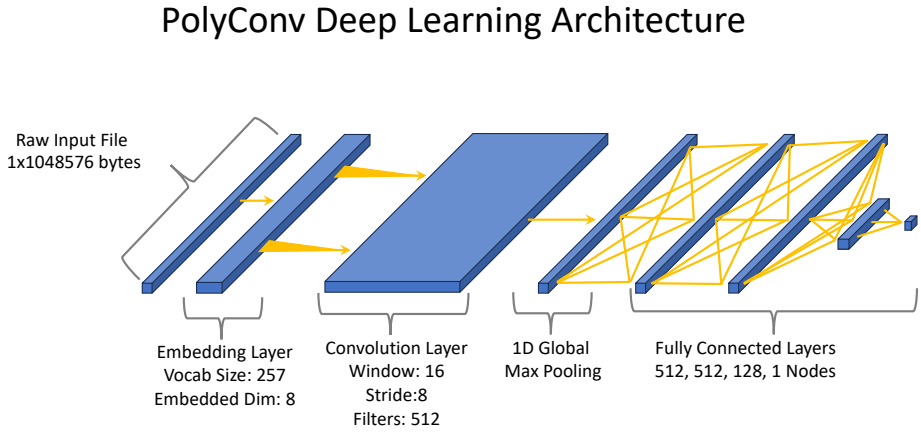


Figure 2.16: PolyConv Architecture

set. PolyConv scored a PR-AUC of 0.99998, the highest score for all the models we evaluated. MalConv—when trained on this novel task—scored a slightly lower PR-AUC of 0.99989, outperforming both PolyCat and CatBoost. The model results are summarized in Figure 2.17.

2.3.6 RQ3: Comparison to Existing Signature-based File-format Identification Tools

This section compares our best-performing polyglot detection model, PolyConv, to existing tools for format identification to determine which approach is best suited to identifying polyglot files and labeling their contents correctly.

Within the context of cybersecurity, there are two complimentary questions of paramount importance: detection and analysis. We trained two versions of our best-performing model, PolyConv, that differ only in the final layer to suit detection and analysis needs.

The first version is a binary classifier (polyglot or monoglot) for use in filtering out polyglots on an endpoint. This is intended for file upload services that only want to allow uploads of known formats, e.g., images.

The second version is a multi-label classifier to identify all of the formats detected within a file. This provides two benefits. First, the labels can be used to route files to all applicable file-format feature extraction or signature-matching routines rather than a single format-specific model or signature subset. This means that the remainder of an existing EDR tool's extraction and detection routines do not need to be altered. Second, the labels provide an analyst with introspection, revealing not only that the file is a polyglot but also which format-specific tools/routines they should use to examine the covert format(s) hidden in the polyglot. This is intended to reduce the response time necessary for secure operation center (SOC) analysts that must handle a high volume of alerts.

2.3.6.1 Tools Tested

We established a baseline for performance by testing existing file-format identification tools on the Wild Polyglots data set: *file* Darwin et al. (2019), *binwalk* ReFirmLabs (2021), *TrID* Pontello (2020), and *polydet* Campion and Laloge' (2018). We also evaluated *polyfile* of Bits (2022), a DARPA-funded tool developed by Trail of Bits for detecting unusual files. Of the aforementioned tools, *file* and *TrID* are well-established signature-based utilities for file-format identification. VirusTotal, a widely used anti-virus aggregator (`www.virustotal.com`), utilizes *TrID* when reporting detected formats. *Binwalk* is a file-carving tool that has been used by analysts to find and extract hidden files. We selected these tool to establish a baseline because of their wide-spread use (*file*), cybersecurity application (*binwalk*, *TrID*), and polyglot-awareness (*polyfile*, *polydet*). We leave as future work a comparison to *FiFTy* Mittal et al. (2020) and *Sceadan* Beebe et al. (2013b), as these detectors do not appear to be polyglot-aware, but might be re-trained in order to properly label polyglot files. Since *file* outputs labels and not probabilities, the precision-recall curve is not an appropriate metric when comparing our deep learning model to existing tools. Instead, we used the F1 score that balances recall and precision. For any cybersecurity system deployable in the real-world, the ability to detect malware/polyglots (recall) must be tempered by a low probability of false positives (precision) to prevent red-flag fatigue. Therefore, we use F1 to provide a balanced evaluation.

2.3.6.1.1 Binary Comparison

Figure 2.18 considers the performance of each tool in a binary context, determining if the tool detects the presence of two or more formats in one file. *TrID* aggressively speculates as to which formats are present in a file, assigning a percent score to each possibility. We therefore omitted the performance of *TrID* as a multi-label detector as this behavior put it at a disadvantage compared to the other tools. As Figure 2.18 demonstrates, none of the existing tools approached the F1 score, precision, or recall

of our PolyConv deep learning model. All of the tools had a relatively high precision and low recall, indicating that false negatives were the primary cause of the low F1 scores.

The recall for *file* was lower than expected as the tool reported multiple formats when examining BMP, EXE, HTA, and PHP monoglots. The EXE false positives may have been caused by the presence of other files embedded as resources. Although it was outperformed by our PolyConv model, *polyfile* was the best binary performer among the existing set of tools by F1 score.

2.3.6.1.2 Multi-label Comparison

Figure 2.19 considers the performance of each tool in a multi-label context where a true positive means the tool correctly identified both the count and the exact formats present in each file. None of the tools performed well compared to the multi-label version of PolyConv.

Of the existing tools, *polydet* outperformed the other tools in all three metrics by a noticeable margin. With regard to the remaining tools, *file*'s precision is unusually low given its widespread use and long development history. Upon examination, we found that *file* did not differentiate between PowerShell and JavaScript files; instead, it applied the generic label of ASCII or Unicode text. This behavior almost exclusively accounted for the lower precision.

The lack of required signatures for script files makes signature-based detection difficult for these script formats. Upon further inspection we found that *polyfile* and *polydet* share *file*'s dependence on Libmagic, which labels PowerShell and JavaScript as either ASCII or Unicode text. While it might seem unfair to expect Libmagic to differentiate between different forms of ASCII or Unicode text, we consider it important for analysts to be aware of this opaque label. A harmless log file of unstructured ASCII text presents a very different level of danger compared to a functional JavaScript file.

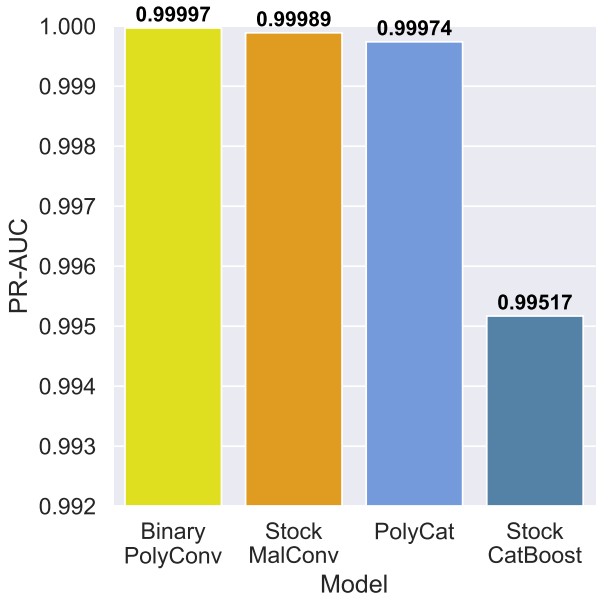


Figure 2.17: Precision-Recall AUC Scores: Our deep learning model, PolyConv, slightly outperformed the stock version of MalConv upon which it is based as well as CatBoost and Polycat.

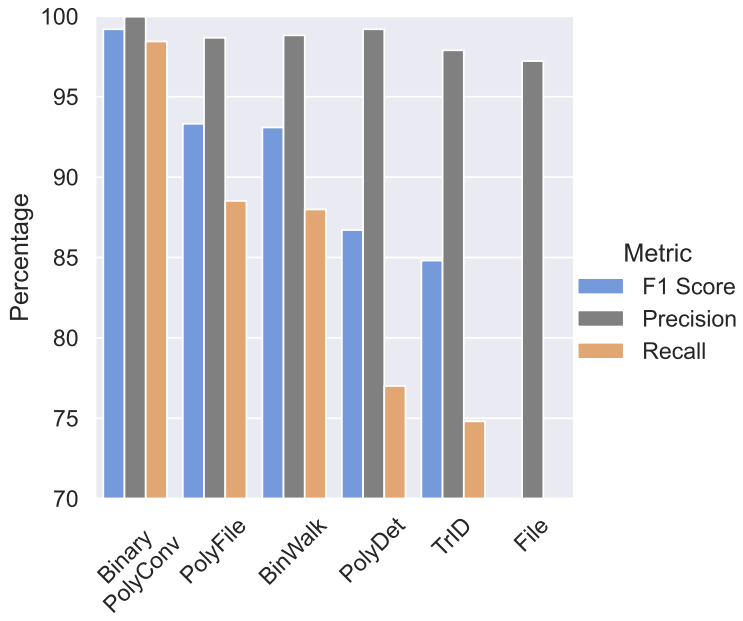


Figure 2.18: Binary Performance vs Existing Tools: PolyConv exceeded the F1 score, precision, and recall of all existing tools by a large margin.

2.3.7 RQ4: Methods for Addressing Image-based Polyglots

Given the prevalence of image-based polyglots in adversary usage and the relative simplicity of image formats, we developed tools for detecting and remediating polyglots that employ an overt image format.

We first tested YARA rules in the hopes that the comment markers/delimiters present in image files would allow for rule-based detection of extraneous content. However, we found that their recall of 82.08% and F1 score of 90.15% were too low to be useful except in situations where high throughput is tantamount. We then turned to the content disarmament and reconstruction approach.

2.3.7.1 *ImSan*, a Content Disarmament and Reconstruction Tool for Image-based Polyglots

Content disarmament and reconstruction (CDR) tools present an alternative approach to the pre-processing filtering approach for which we have provided solutions. CDR tools allow an end user to strip all but the most trustworthy content from certain formats. Where highly flexible formats, like PDF, have proliferated, these tools have emerged to provide secure use of files that abuse the format flexibility.

Although we have not exhaustively examined this approach, we have developed an image sanitization tool to demonstrate the potential of CDR in disarming polyglots. Our tool, *ImSan*, disarms image-based polyglots by stripping away all file contents that are not required to display the image. The process is quite straightforward:

1. The image file is loaded into Pillow, a fork of the Python Imaging Library
2. The image contents are then written to a new file with the option to strip all metadata activated
3. The new image file has no extraneous content before/after the image contents (*stack*/*cavity* polyglot) or inserted into comment areas (*parasite*/*zipper* polyglot)

ImSan can disarm any of the formats that are fully supported (read/write) by Pillow: BLP, BMP, DDS, DIB, EPS, GIF, ICNS, ICO, IM, JPEG, JPEG 2000, MSP, PCX, PNG, PPM, SGI,SPIDER, TGA, TIFF, WebP, XBM. Note, *ImSan* should be run in an isolated environment to ensure that no vulnerability in Pillow (2 CVE's reported in 2022) could allow a malicious image to gain execution when the image is parsed.

ImSan disarmed 100% of the image polyglots in a subset (n=392) of image polyglots drawn randomly from the benign Wild Polyglots data set. A small subset was chosen so we could manually verify disarmament through visual inspection of the image's code rather than relying on one of our detectors. An evaluation of commercial CDR tools against polyglots (including those that are not image based) and the potential methods of circumventing CDR solutions, while out of scope for this work, would be a valuable direction for future work to explore.

2.3.8 Discussion

2.3.8.1 Contribution Summary

We presented the first, to our knowledge, survey of polyglot usage by malicious actors in the wild, demonstrating that polyglot files are an actively used TTP by well-known malicious actors, answering **RQ1**.

In order to answer **RQ2-RQ4**, we created a novel data set of polyglot and monoglot files based on file formats and file-combination methods utilized by malicious actors in the wild.

Using this Wild Polyglots data set, we evaluated a number of different machine learning models before focusing on the top two performers, PolyConv and PolyCat. we improved these two models via alterations to their architecture and feature space, respectively.

We found that PolyConv, in both binary and multi-label versions, was effective at detecting polyglots and correctly labeling their contents **RQ2**, providing analysts with a tool to detect, reroute, and investigate potential polyglots.

PolyConv only slightly outperformed the model upon which it was based, MalConv, demonstrating that MalConv effectively learned to distinguish between polyglot and monoglot files when trained on this objective, despite being designed to detect malware. This is a novel use of MalConv considering that the model was designed to detect PE malware.

Based on our experiments, the improvement from MalConv to PolyConv was due to the reduction of the window and stride size as well as increasing the number of filters and layers. We theorize that the much smaller window/stride allowed the model to learn filters that register even small areas of code with a distinct byte pattern. The need for more filters may be due to the wide variety of formats, each with their own distribution of unique byte patterns, upon which we trained. On the other hand, removing the attention and gating mechanism did not reduce the model's classification performance.

We answered **RQ3** by demonstrating that existing tools do not reliably detect polyglot files, even when designed with an awareness of polyglot files.

To answer **RQ4**, we produced a set of YARA rules for detecting extraneous content in image files, but found their performance lacking. The rules are available upon request. We then created *ImSan*, a content disarmament and reconstruction tool that sanitizes image files, demonstrating that it disarmed all of the image-based polyglots with which we tested it.

2.3.8.2 Limitations

We cannot guarantee that our deep learning models will perform well on polyglots formed from file formats not included in the training data. File formats based on Open XML (Microsoft Office) are a common malware vector that have not yet been

thoroughly explored as polyglot components. PolyConv is format agnostic so we hope that this model's release will prompt further research using additional file formats.

File size is a limiting factor for malware classification models trained on raw bytes rather than extracted features. Detection could be evaded by utilizing a polyglot whose overt format exceeds the full input capacity of the model, meaning the covert format would not be ingested. We tested head and tail scanning on our data set, but found that this did not improve results since the vast majority of our data set is within the maximum capacity of our PolyConv model. Head and tail scanning could still be evaded if an adversary inserted the second file near the middle of a particularly large first file or appended a large amount of data after the second file contents.

We also tested the YARA rule approach, but found it A) limited by the need to write novel rules for each possible combination of file formats and B) the lack of required signatures in many flexible file formats.

2.3.8.3 Future Work

Given the ubiquity of signature-based tools, the development and demonstration of a set of signatures or rules that detect a very high percentage of functional script files as well as the ability to differentiate between different scripting languages would be valuable. While JavaScript (JS) lacks a strict signature, it might be possible to craft a set of rules, perhaps using a tool such as AutoYARA Raff et al. (2020b), that detects all JS files that have enough control flow instructions to be functional/malicious. This would require the collection of a large number of files for each language, along with some level of assurance that the data set is extremely diverse with respect to the file contents/functionality. If Libmagic were able to reliably distinguish JavaScript and Powershell files from other forms of text, the performance of *polyfile* and *polydet* could improve markedly. We are not confident the rule-based approach is tractable, yet cannot rule it out.

Since PolyConv utilizes a global max pooling layer, it is translation invariant. That said, a demonstration of its ability to generalize to novel insertion areas remains

future work. We consider translation invariance an important feature in order to future-proof a polyglot detector. Given the flexibility in file formats, it is possible that novel polyglot creation methods will emerge in the future that hide the second file in a novel area of the first file. Therefore, a demonstration that PolyConv is resilient in the face of novel combination methods would demonstrate that future models for polyglot detection should also be translation invariant.

Future work should include the implementation of an intelligent method for subselecting or compressing large input files so they fit within the maximum capacity of a model trained on raw bytes. Head and tail scanning would catch data appended to the very end of the file, but could be evaded by inserting data earlier in the file or appending more benign content after the additional malicious content. Therefore, a more robust input reduction method should not follow a fixed pattern such as always scanning N bytes from the head and M bytes from the tail. Such a method may exist in other domains; we look forward to developments in this area.

Finally, PolyConv needs to be trained and tested on the same wide variety of files as the ubiquitous *file* utility in order to see widespread adoption.

2.3.8.4 Recommendations

2.3.8.4.1 Polyglot Detection

Deep learning clearly outperformed other approaches to polyglot detection. Therefore, EDR tools or file upload services concerned with handling polyglots should invest in deep learning to identify and analyze incoming files. Our work demonstrates that the approach is effective and tractable. CDR tools should be used to strip extraneous content from files without loss of (benign) functionality, assuming said tool is fully compatible with the overt formats of incoming files.

CDR tools should become ubiquitous for the formats where which which they are compatible. As for whether filtration or disarmament is the better approach, we would argue that defense-in-depth is the best approach. Enterprises should use commercial

or open-source CDR tools on all file formats that said tools can successfully disarm and reconstruct reliably. For formats that are more complex or difficult to reconstruct reliably, enterprises should train PolyConv to catch any polyglots exploiting these formats. We suggest it is more tractable to train a detector to recognize a wide variety of byte patterns than it is to code reconstruction routines for all of the necessary file formats.

2.3.8.4.2 Existing Signature-based Tools

Until deep learning models trained on the huge variety of files necessary for enterprise-scale use are available, we recommend utilizing *polyfile* to filter suspected polyglots. For routing polyglots to multiple models or gaining introspection for analysis, we recommend using *polydet*. Although neither of these tools were nearly as effective as our PolyConv model, they were the top performers among existing tools. Users should bare in mind that they do not discriminate between JavaScript, PowerShell, and plain text.

2.3.8.4.3 File-format Specifications

File-format specifications and their practical implementations determine whether or not a polyglot can be created from two or more file formats. One way to continue to use signature-based detection would be to require all file formats to use a signature at the same fixed offset. Since two unique signatures cannot exist at the same offset, any combinations of file formats in a single file would not be valid/functional.

This solution do not appear practical as it would break backward compatibility for an enormous variety of files, e.g., Zip and DICOM formats which are intentionally flexible in this respect. The difficulty in securing flexible formats while preserving backward compatibility with existing files led to the development of CDR tools in the first place.

Emerging file-format specifications, however, can benefit from our lessons learned. The chief difficulty lies in preventing a flexible format from being inserted into or

appended to the novel file format. Toward preventing this, we recommend the following specifications as a starting point:

- Digitally certify all files upon creation
- Require that officially supported parsing programs verify this certificate
- Ensure the certification area does not contain slack space that is ignored during the hash calculation
- Specify a file-format signature that must be present at offset zero
- Do not provide optional/slack spaces where extraneous data could be hidden
- Any "reserved for future use" fields should be validated to ensure they do not contain unexpected values
- Validate that any padding areas (due to memory paging) do not contain unexpected values

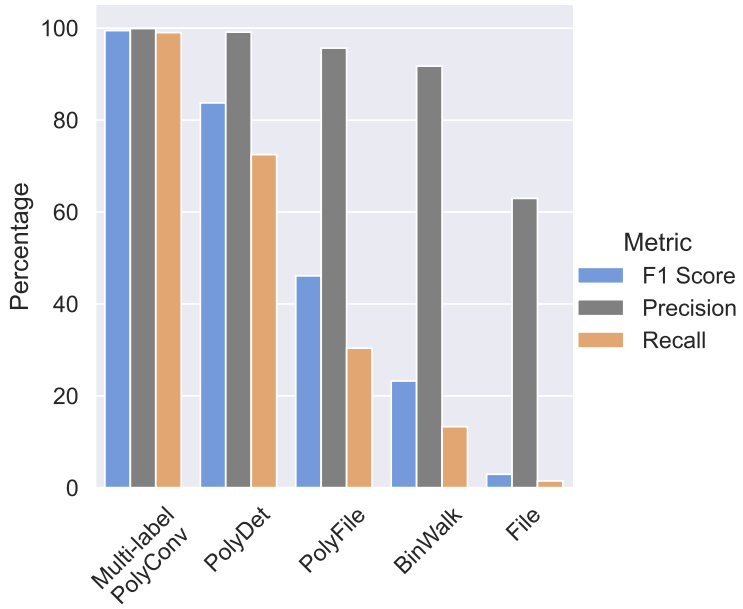


Figure 2.19: Multi-label Performance vs Existing Tools: PolyConv also proved more adept at correctly identifying all of the formats contained within a file. Of the existing tools, *polydet* provided the most reliable file-format identification.

Chapter 3

Unguided Static Binary Instrumentation

This chapter details our initial experiments with binary instrumentation actions. These methods were applied without any optimization from an AI agent and tested against a machine learning-based detector and a signature-based detector. The methods were effective at evading both types of detectors, prompting further research into static binary instrumentation. This chapter provides the initial impetus and justification for our interest in developing more advanced static binary instrumentation.

3.1 Five Methods for Instrumenting PE files

Static binary instrumentation can be effective at evading detection without any optimization from a machine learning model. This chapter details how Windows PE files can be modified and reports how effective these modification are on two COTS malware detectors, one of which is ML-based while the other is signature-based. The instrumentation/perturbation/alteration/transformation techniques developed below were written in Python to alter Windows PE files.

This brute force approach applies a single alteration to each malware file. These alterations include appending bytes to the end of a file Kolosnjaji et al. (2018a), adding a section header to the section table, adding a section header with corresponding data appended to the end of the file Anderson et al. (2018), altering nonessential header informationAnderson et al. (2018), flipping bits in various sections, and code caving.

Append Bytes: Appending bytes to the end of a file is a straightforward method. Bytes can be appended by using the *cat* utility. A Windows PE is tolerant of data added to the end of the file that is not loaded into virtual memory. This area is known as the overlay. For our tests, we developed several basic variants of append so that padding zeros, random data, or strings from a file can all be added to a malware file. Our objective here was to determine which perturbation is most effective for shifting classification.

Add Section Header: This method adds a new section entry to the section table, but does not add any actual section contents to the file. Section header addition takes advantage of the padding zeros normally found at the end of the section table of a PE file. PE files follow a strict format that tells the operating system how to allocate memory for the program, as well as where to find the address of the entry point. Important information contained in this section includes the number of sections, the sizes of various sections of the file, as well as the flags for each section. Each section that comprises the PE file has a header with data on that section's raw and virtual sizes, in addition to the flags. Padding zeros are left at the end of the section headers, often allowing enough room for another section header to be added. Flags, raw size, and virtual size all have to be calculated and encoded. Encoding these values so that the new data looks authentic required extensive research into how values for each field are calculated. For instance, virtual memory is allocated in pages, so virtual size has to be added in units of 200kb (Windows default) at a time. Raw size, on the other hand, is based on the actual size of code as written in the file. Our objective is to make the new section header look as authentic as possible to an endpoint detector.

Append with Header: This perturbation is a combination of the two previous methods. Bytes, along with a matching section header, are inserted into a file. In cases where the models being tested ignore data at the end of a file, we hope that the added section header will add weight to the additional data. The objective is to add a large amount of benign data to counteract the negative score of the existing malicious code. We will also test whether changing the flags on the appended section will affect how much attention the appended date gets. If a model under evaluation ignores data in read-only section, perhaps making the appended data writable or executable will compound the positive effect of the benign data upon classification. This technique was by far the most complex to write and required a great deal of research into the structure of PE files. It scans through a target file to find the padding area at the end of the section headers, then writes appropriate values for a new section into this space. In addition to allowing the user to set flags for the new section, this technique also provides new virtual and raw sizes for the appended section based on the amount of data appended at run time while still conforming to conventional Windows values for these sizes. The raw pointer to this section is also calculated.

Rename Headers: Section names in the section table are ignored by the program loader. Altering nonessential fields is a counter-intuitive approach. One might assume that any model would be intelligent enough to ignore nonessential data like section names. However, this line of thought athropomorphizes the ML model to an excessive degree. According to Anderson Anderson et al. (2018), et al., renaming section headers proved effective at masking ransomware from detection. We want to determine whether or not this minimalist approach is still effective against thoroughly-trained production models rather than research models trained on small data sets.

Byte flip: Byte flipping is another minimalist approach. The goal here is to disrupt the hash value while searching for significant bytes. Although the original idea was to flip individual bits, we opted to flip eight bits - one byte - per step. In practice this amounts to randomizing the value of a byte, but we stuck with term bit flip for convenience's sake. While a single bit flip would be enough to counter a regular hash,

we wrote scripts to flip bits throughout the file to defeat rolling hashes as well. As with the other scripts, we have the option to choose, based on section flags, where to flip bits to learn which sections the model emphasizes. We are also able to choose between flipping bits in written code or in padding zeros. The former option carries a risk of breaking the file. The latter option does not disrupt the file, but may fail to be significant enough of a perturbation. Thus we can gradually increase the scale of our perturbation. We could start with a single flip of eight bits in the padding zeros, then begin flipping more and more bits throughout the file. If this approach is still insufficient to alter classification, we will begin flipping bits in written code, checking after each flip to see if we have broken the file.

Code Caving: The last perturbation is code caving. This technique first scans a binary file for consecutive zero bytes. If any strings over an arbitrary length are found, the perturbation technique then writes data to these zero bytes. This technique allows for additional data to be written into the padding bytes and unfilled memory segments of a binary file. If applied correctly, this technique does not damage the file. It is important to note that the caves found and filled by this technique are not reachable by code execution. The operating system still perceives these areas as unreachable padding bytes or unused memory. It is possible to make these caves reachable. but this is a more advanced technique that we have not developed yet. For our purposes the code caves allow us to inject benign data into malware files in the hopes of confusing classification. This technique differs from bit flipping since it only operates within available caves and allows for the selection of data to be added.

3.2 Instrumentation Variations

There are 74 total variations on the five principal techniques. Techniques that inject data have multiple source options, and nearly all of the techniques can be aimed at specific sections of a target file. This targeting allows us to learn which sections and descriptors the ML model considers relevant to classification. In the future, each of

these variations could be provided to a guidance model as a separate action. For the time being, the perturbation variations allow for a data-driven exploration of the action space afforded by adversarial samples.

- Technique: append data with corresponding section header
  - Flags: zero, random, source
    - Zero: appends all zeros to end of file
    - Random: appends randomized data to end of file
    - Source: appends data from source file to end of file, requires amount in Mibibytes
  - Flags: executable, read-only, read-write, disposable
    - Executable: Sets new section permission as executable
    - Read-Only: Sets new section permission as read only
    - Read-Write: Sets new section permission as read and write
    - Disposable: Sets new section permission as disposable
  - Flags: gibibytes: changes scale of new data to gibibytes from mibibytes
  - Variations: 12 total possible
- Technique: byte flipping
  - Flags: single, count, random
    - Single: randomize one byte (equivalent to flipping eight bits) in the specified sections
    - Count: randomize the specified number of bytes in the specified sections (loops around if specified count exceed byte count in specified sections)
    - Random: randomize a random number of bytes, capped at five percent of specified sections
  - Flags: executable, read-write, disposable, whole file

        * Executable: bit flips will occur only in executable sections of the file
        * Non-Executable: bit flips will occur only in sections not marked as executable
        * Disposable: bit flips will occur only in sections marked as disposable
        * Whole File: bit flips can occur in any section
    - Flags: Massive: removes upper bound on random percentage
    - Flags: padding, non-padding, all
        * Padding: bit flips will occur only in the padding bytes of specified sections
        * Non-Padding: bit flips will occur everywhere except the padding bytes of specified sections
        * All: bit flips will occur in all areas of specified sections
    - Variations: 36 total possible
- Technique: only append data
    - Flags: zero, random, source
        * Zero: simply appends zeros to end of file
        * Random: simply appends random data to end of file
        * Source: simply appends data from specified file to end of file
    - Flags: gibibytes: changes scale of new data to gibibytes from mibibytes
    - Variations: 3 total
- Technique: code caving
    - Flags: file source, random source
        * File Source: injects code from a specified file source
        * Random Source: injects randomized code

  - Flags: executable, non-executable, disposable, all sections, whole file
    * Executable: only injects code into caves in executable sections
    * Non-Executable: only injects code into caves in sections not marked as executable
    * Disposable: only injects code into sections marked as disposable
    * All Sections: injects code into all sections
    * Whole File: injects code into header areas as well as all sections
  - Variations: 10 total

- Technique: rename headers
  - Section Flags: text,rdata,bss,rsrc,idata,edata,debug,data,buildid,xdata
  - Other Flags: random,random numeric, unexpected
    * Text: This renames all sections to .text, which typically corresponds to executable code
    * Rdata: This renames all sections to .rdata, which typically corresponds to read only data
    * Bss: This renames all sections to .bss, which also typically corresponds to read-write data
    * Rsrc: This renames all sections to .rsrc, which typically corresponds to resource data
    * Idata: This renames all sections to .idata, which typically corresponds to import data
    * Edata: This renames all sections to .edata, which typically corresponds to export data
    * Debug: This renames all sections to .debug, which typically corresponds to debug data

    - Data: This renames all sections to .data, which typically corresponds to read-only data
    - Buildid: This renames all sections to .buildid, which typically contains a unique data string from the compiler
    - Xdata: This renames all sections to .xdata, which typically contains exception data
    - Random: This randomly chooses from the other name options
    - Random Numeric: randomly chooses from common numeric names for sections that were extracted from sample binaries
    - Unexpected: renames all sections to .benign for obvious reasons
  - Variations: 13 total

3.3 Evaluation Scenario

The perturbation techniques thus far developed were applied to malware samples from an experiment conducted on behalf of an unnamed ORNL sponsor. The evaluation took place on a realistic cyber range built by ORNL specifically to test several ML-driven malware detectors on behalf of the sponsor. The objective was to determine which, if any, of the systems tested was mature enough to detect a wide range of malware. Critically, the endpoint detectors had to function without access to malware signature updates from a command network. The range is composed of several servers running on an isolated and firewalled network. Each server is dedicated to hosting virtual machines (VMs). Each VM is spooled up with either a Windows or Unix operating system. A vendor's endpoint detector is also installed on each VM. Native malware detection is disabled so that only the endpoint can detect incoming malware. During each test run, malware samples are uploaded to each VM. The orchestrator then pauses for a pre-ordained period before collecting any alerts issued by the vendor

detector. The malware samples came from from a variety of public and private sources. Special effort was taken to find the most potent samples available.

ORNL tested five state-of-the-art ML-based malware detectors against 200,000 malware files. The computational time required for this project was calculated at 86 years before orchestration software was used to distribute the processor time to the virtual machines. The victory criteria was the minimization of a function that incorporated the cost of operation as well as the estimated cost of recovery from all intrusions.

With regards to the binary instrumentation project, this challenge provided a large set of malware files that were correctly classified as malware by one or more classifiers. Moreover, ORNL staff was able to record exactly which files were correctly classified by which detection system.

3.4 Results of Current Perturbations

In this experiment a number of perturbations were selected and applied to correctly classified malware from the sponsored experiment. These altered malware samples were then presented to two malware detectors. These detectors were in use by large enterprises and considered state-of-the-art. One relied on signatures for detection while the other made use of machine learning. Their names are withheld per an agreement. This section will demonstrate that the perturbation techniques already developed are strong enough to alter the ML and signature-based classifiers without the benefit of guidance from our adversarial neural network. The results will provide insights for the next generation of perturbations. They will also allow us to to track the effects of guidance as opposed to perturbation once algorithmic optimization is attached.

We extracted 15,000 malware samples from the sponsored experiment. All of these malware samples were correctly classified by both the signature-based detector and the ML-based detector. Next we selected our perturbations. Recall that we had

completed 74 variations at this point in time. Since this test run is a demonstration of vulnerability rather than an exhaustive study, we used intuition to select representative variations. This intuition was based on a synthesis of research performed during the first six months of the project. The author selected 24 variations based on the five basic perturbations. The objective was to adequately represent the 74 variations developed while still maximizing the possibility of evading detection. The variations selected are as follows:

- Append 4MiB of random data to end of target file
- Append 4MiB of strings extracted from video game install file to end of target file*
- Append 4MiB of data from video game install file to end of target file*
- Apppend 4MiB of zeros to end of target file
- Inject random data into caves in whole file, including header area
- Inject random data into caves in all sections of target file
- Inject random data into caves in executable sections of target file
- Inject random data into caves in non-executable sections of target file
- Inject random data into caves in disposable sections of target file
- Append 4MiB of random data to end of target file with custom header marking new section as executable
- Append 4MiB of random data to end of target file with custom header marking new section as read only
- Append 4MiB of random data to end of target file with custom header marking new section as read or write

- Append 4MiB of random data to end of target file with custom header marking new section as disposable
- flip one byte (eight bits) in padding bytes of any section of target file**
- flip one byte (eight bits) in padding bytes of executable sections of target file**
- flip one byte (eight bits) in padding bytes of non executable sections of target file**
- flip one byte (eight bits) in padding bytes of disposable sections of target file**
- Rename all headers of target file to .text
- Rename all headers of target file to random numeric name
- Rename all headers of target file to .benign
- Rename all headers of target file to .data
- Rename all headers of target file to .bss
- Rename all headers of target file to .buildid
- Rename all headers of target file to .debug

* These two append methods depend on external files for source material. Based on Skylight Cyber's bypass of Cylance Cyber (2019) (a deep learning-based malware detector), the author chose an install file from a video game for source material. Two variations were selected. One appended straight from the install file, while the other appended from an ASCII strings-only file extracted from the install file. The author was curious whether strings like Skylight utilized were more or less advantageous than the rest of the binary contents.

* While flipping random bits would obviously run a high risk of breaking a file, we avoid this issue by only flipping padding bits during this test run. Expected evasion is low considering how unimportant these bits are, but we prefer functionality to evasion.

In total, 353,251 new malware samples were created from the original 15,000 by applying these 24 perturbation methods. Some methods were successful on all malware samples, while others only succeeded on a fraction of them. The 353K files were divided into three approximately equal subsets. Since the orchestrator operates one vendor per run, this meant we had to make six runs, three for each of the two detectors. To prepare the files for these runs the author wrote scripts that generated the new malware by applying the selected perturbations *en masse* to the selected corpus. Next, the author generated a csv file with relevant identification for each new malware file. This file was then uploaded to the orchestrator along with an OS image and a pre-configured version of the vendor detector system. Once alert detection was carefully set up, the tests each ran for a minimum of fifteen hours.

Our methodology provides specific information on the effectiveness of each action in isolation. We know exactly which files evaded which detector as well as the aggreggate evasion rate for each variation. We also have copies of each malware file before and after perturbation for further study.

3.5 Implications of Current Research

The overall results of the test answered two critical questions. Yes, a production-grade Ml-based detector is vulnerable to even the simplest perturbation. Yes, more complex perturbations are much more successful at evasion. Thus our research is validated on two fronts. More importantly, our preliminary results show that further research into more complex perturbations and guidance systems is more than warranted. The ML-based detector suffered double the evasions of the signature-based competitor overall, and displayed a significant vulnerability to the more complex perturbations. We will now review the results in detail.

By a large margin, appending data combined with caving a new section header had the highest evasion rate. Of these append w/ header variations, marking the new section as executable achieved the highest evasion rate at 52% against ML and

26% against Signature. Appending random data alone achieved an evasion rate of 12% against ML and 10% against Signature. Since altering header information is responsible for the large change in evasion rate, we have some evidence that the ML model's internal model places significant emphasis on header data. The success of these variations provides ample evidence that moving beyond simple appends and bit flips and beginning to alter the structure of a PE itself will yield important results.

Code caves achieved the second highest evasion rate (21-39%) against the ML detector. As intuition would suggest, the evasion rate of caving appeared to increase according to the importance of the sections where the caving took place. Disposable sections scored low, non-executable sections scored in the middle, and executable sections score the highest of the section caves. Why caving in the file sections scored higher that caving throughout the whole file is curious.

Simply appending to the end of a file was surprisingly influential. After all, this perturbation does nothing except add inaccessible data to the end of a binary file. A success rate of 8-18% was not expected. This indicates that the ML model places significant weight on the malware file as a string of bytes rather than a functional unit. Functional analysis is far more robust, but difficult to integrate into a production ML, based on industry conversations. Treating a malware sample as a string of bytes, however, is very straightforward since it is almost identical to image classification. As our tests demonstrate, however, it leaves the model vulnerable to shallow perturbations that barely alter the file. Similarly, the fact that flipping a single set of eight bits in the padding bytes of a disposable section evaded ML-based detection 17% of the time is somewhat alarming. Since that variation completely failed to bypass the signature-based detector, this is a clear cut instance of a manual methodology defeating the ML model. This leads the author to theorize that implementing a rolling hash in the context of an ML model is not a straight forward proposition.

A curious byproduct of this test is the vulnerability the signature detector demonstrated to header renaming. In this area the signature detector (15-17% evasion) was just as vulnerable as the ML model (5-21% evasion). Since signature detection

performed remarkably well against bit flipping (0-2%), the author believes that the signature-based detector had specific defenses in place for bit flips but not for header renaming. The signature detector was also surprisingly vulnerable to appending with header caving (26%). These vulnerabilities highlights the frustrating nature of cyber defense. Novel, albeit simple, approaches are a constant threat to static defenses. In some cases, vulnerabilities that seem too simple to be effective to the research community still pose a challenge for industry tools.

In sum, the preliminary results indicate that vulnerabilities exist to even basic perturbations. Previous research had indicated that these methods could be effective against a production ML. We have demonstrated that they are capable of challenging a production-grade ML-based detector. Given this detectors vulnerability to basic perturbation, it is clear that this vendor analyzes malware more as a string of bytes than as a series of function calls. This emphasis provides an adversary with a large attack surface since nonfunctional bytes can be added to confound classification. Moreover, the significance placed on the section header provides a specific, high-value target area.

Appending and flipping performed better than the nil results expected, but it was code caving and appending with headers that achieved the most interesting results. More importantly, the higher evasion rates of the more complex perturbations indicate that we are only beginning to explore the depths of opportunity afforded by adversarial samples. Appending with header caving slightly alters the structure of a PE file. We believe that perturbations that further alter the structure and/or function of PE files will yield further results. To that end, the next section describes some of the methods we will investigate next.

3.6 Future Perturbation Techniques

Techniques that are still undergoing investigation include PE partitioning, expanding code caving Kreuk et al. (2018) into ROP injection Fleshman et al. (2018), IAT

Hooking Rosenberg et al. (2018), selective packing, and code metamorphosis. These techniques are more advanced and run the risk of altering functionality if not carefully applied.

Based on the preliminary results developed above, the author intends to, first and foremost, explore methods for partitioning PE files into their smallest manipulable chunks. Zombie, a well-known malware author from Russia, once authored a piece of software known as the Mistfall Engine Ferrie and Szö (2001). This malware delivery mechanism functioned by partitioning a benign PE file into its smallest manipulable chunks. These chunks could then be safely rearranged, allowing malicious code chunks to be woven into the benign file. The benign file was then reconstructed with its malware parasite tucked safely inside. Incredibly, Zombie managed to complete this process without altering the functionality of the benign file. The author believes that these techniques could be reproduced within AMIGO to enable comprehensive alterations to malware files. The engine is publicly available and will be studied in the near future.

Code caving involves writing code to padding zeros wherever possible. This code is not accessible for execution, it merely adds noise to the file. A further development of code caving is the use of ROP injection for perturbation. This advanced technique involves a buffer overflow that allows the program to redirect execution to normally inaccessible code. This technique can be used to hide code segments flagged as malicious in code caves that are only reached via ROP. This technique requires sufficient room in the file as well as the overflow attack.

IAT Hooking is the redirection of control flow via the Import Address Table. PE files use the IAT to access external functions. This flow can be hijacked by changing the addresses stored in the IAT so they point toward new functions. If these new functions conclude with jumps to the original external destinations, they are called pass-through functions. These do not affect functionality when written carefully.

Code metamorphosis is one of the more extensive techniques. A metamorphic engine combs through the malware file, swapping functions with equivalent code. For

instance, a 'for' loop might be swapped for a 'while' loop. This process is inherently complex, as these rewrites shift code up and down in memory. The engine has to account for each shift and calculate new offsets for jumps to and from the shifted area. As with the ROP Injection, this technique allows for a great deal of perturbation, but can very easily damage the file. These complex perturbations make the functional analysis a necessity.

Table 3.1: Evasion Rate of Unguided Perturbations against COTS Tools

Perturbation Variation	Evaded ML Detector	Evaded Signature Detector
Overall	22.53%	13.58%
Header Rename .text	5%	15%
Append Strings	8%	8%
Header Rename Numeric	9%	16%
Bit Flip Non-Exe Sections Padding	10%	1%
Header Rename .benign	11%	16%
Append Random	12%	10%
Bit Flip Whole File Padding	13%	2%
Header Rename .data	14%	15%
Bit Flip Exe Sections Padding	15%	2%
Append File	16%	10%
Bit Flip Disp Sections Padding	17%	0%
Header Rename .bss	17%	16%
Header Rename .debug	18%	16%
Append Zeros	18%	10%
Header Rename .buildid	21%	17%
Cave Random Disp Sections	21%	2%
Cave Random Non-Exe Sections	25%	14%
Cave Random Whole Files	26%	16%
Cave Random Exe Sections	34%	8%
Cave Random All Sections	39%	17%
Append w/ Header Read/Write	46%	26%
Append w/ Header Disp	47%	26%
Append w/ Header Read	47%	26%
Append w/ Header Exe	52%	26%

Chapter 4

Survey of Adversarial Binary Methods

This chapter consists of a wide-ranging survey of methods for creating adversarial binaries by combining AI guidance/optimzation with a set of static binary instrumentation actions. The survey covers reinforcement learning (RL), genetic algorithm, and gradient-based agents and focuses on the implementation of binary instrumentation actions. This survey revealed that functionality preservation is rarely given sufficient consideration and that the RL-based papers use comparatively trivial perturbations. These two observations motivated our final experiment which sought to improve an agent's performance by creating more effective actions that have been thoroughly tested for functionality preservation.

4.1 Introduction

Malware detection, a core component of cybersecurity, must constantly evolve to identify new threats. This evolution is driven by the discovery of new methods for evading detection. As artificial intelligence methods make gains in a wide variety of fields, it is only natural to see them applied to the evolution of malicious executable files. However, this domain of binary instrumentation imposes a number of unique challenges not present in the image domain where adversarial machine learning first emerged. The challenge of **altering malware without damaging functionality** is what makes

this vein of research **uniquely challenging**. In the image domain, pixels need not be an exact match in order to preserve the correct class label; malware binaries, however, can be fatally damaged (prevented from executing) by the incorrect alteration of a single byte. This **domain constraint** largely **motivated the use of reinforcement learning and genetic algorithms** as opposed to more efficient gradient-based methods. The development of novel binary instrumentation methods appears to be the chief limiting factor of research in this vein. Toward this objective, we recommend the adoption of standardized and comprehensive criteria for evaluating functionality rather than relying upon assumptions drawn from file format specifications.

4.1.1 Criteria and Contribution

The criteria for including papers in this survey was the use of an ML algorithm to instrument a binary file in the input rather than feature space. The domain constraints imposed by working in the input space, especially on binary files, have given rise to the family of static binary instrumentation methods that distinguish this vein of research from other adversarial domains, e.g., images. Note, we deviate from this rule for the paper by Zhang et al. Zhang et al. (2022) because their targeting of a graph neural network trained on the control flow graph presents an opportunity to discuss target models that are not vulnerable to the bulk of instrumentation methods.

Prior surveys in this space focus either on one particular set of methods Demetrio et al. (2021c), the defensive use of reinforcement learning (RL) Nguyen and Reddi (2023) or do not delve deeply into the binary instrumentation actions used by the offensive RL agent Sewak et al. (2022). The survey by Aryal Aryal et al. (2022), while excellent, is a broad spectrum analysis that briefly discusses several file formats and associated methods. In contrast, this survey focuses specifically on Windows PE files, providing a deeper description of the file format's challenges, the implementation details of PE-altering actions, a consolidated taxonomy for said actions, the limitations of existing functionality preservation, and concrete areas where novel actions/experiments

will advance our knowledge of the threat posed by adversarial binaries. This survey will further argue that the composition of standardized malware datasets with known behavior and more rigorous evaluation criteria for functionality preservation and, therefore, the label of adversarial binary, are keys areas where research should be advanced. Note, this survey was prepared as part of an independent study at the University of Tennessee under the guidance of Dr. Edmon Begoli.

4.1.2 Survey Layout

We first define the term *adversarial binaries*, then present the unique domain constraints of binary instrumentation. This section includes includes a comprehensive primer on the most commonly attacked binary file-format, Windows PE, since it's structure gives rise to many domain constraints. Next, we discuss common terminology static vs dynamic malware detection and the use of dynamic detection as an oracle for static obfuscation methods. We then review data science terminology employed in this domain to benefit researchers from other disciplines. The background section concludes with a discussion of the most prevalent algorithms for crafting adversarial binaries along with the rationale, advantages, and disadvantages associated with each family of methods.

The methods section first discusses target models and their features sets before presenting a summary of surveyed papers. Since there is little crossover between papers that utilize different guidance mechanisms, we organize the papers by their guidance mechanism. To facilitate future development of novel actions, the binary instrumentation methods from surveyed works are collected and discussed in detail in Section 4.3.5. Although our goal was an exhaustive compilation, we found sufficient clarity lacking in some papers and have only included actions whose implementations were clearly detailed. The following outline provides a quick reference of the main sections within the survey.

- Introduction

- Background
  - Section 4.1.1: Criteria and Contribution
  - Section 4.2.1: Definition of *adversarial binary*
  - Section 4.2.2: Domain constraints imposed by binary format files
  - Section 4.2.3: Structure of most common target binary (Windows PE)
  - Section 4.2.4: General Malware Detection
    - Section 4.2.4.2: Static Malware Detection
    - Section 4.2.4.1: Dynamic Malware Detection
  - Section 4.2.5: Experiment Design Terminology
    - Section 4.2.5.1: Attack Scenarios
    - Section4.2.5.2: Surrogate Models
    - Section 4.2.5.3: Transferability
  - Section 4.2.6: Optimization Algorithms
    - Section 4.2.6.1: Algorithm Selection Considerations
    - Section 4.2.6.2: Gradient-based Methods
    - Section 4.2.6.3: Genetic Algorithms
    - Section 4.2.6.4: Reinforcement Learning
- Methods
  - Section 4.3.1: Target Models and Feature Sets
  - Section 4.3.2: Gradient and Genetic Papers
  - Section 4.3.4: Reinforcement Learning Papers
  - Section 4.3.5: Instrumentation Action Database
- Discussion
  - Section 4.4.1: Current Limitations
  - Section 4.4.2: Future Work Recommendations

4.2 Background

4.2.1 Definition of Adversarial Binary

According to Szegedy Szegedy et al. (2013), an adversarial sample is found by perturbing an original sample so that it receives an incorrect label from a target model without losing the original sample's true label. A perturbation is a small (bound or minimized in some way) transformation or alteration. The bound is the result of an attempt to maintain the true label as applied by an oracle (the human eye when discussing images). For a malware binary file, the true label is applied based on activity rather than appearance. Therefore, Demetrio defines the specific case of adversarial malware as malware that has been transformed to achieve evasion "without disrupting its original behavior" Demetrio et al. (2021a). The perturbation is bound, therefore, not by the effect on the appearance of the file as it sits on disk or in virtual memory, but by the effect on behavior. Given the large search space (all possible byte values for a binary file of variable length), this altered sample is found or generated using an AI system that samples and scores the search space using an optimization function rather attempting to exhaustively test all possible combinations of byte values in a file that may be several gigabytes in size. According to the U.S. Dept. of Homeland Security, methods that use AI guidance to deceive a target system (the target may or may not be AI-based) are defined as adversarial methods Kusnezov et al. (2023) as opposed to counter-AI methods which deceive AI-based target systems with or without using an AI of their own. The context of our survey is therefore bound to systems that seek to deceive malware classification systems (which may or may not use AI) by training an AI agent to alter binary files without loss of function. Although adversarial methods originally used gradient-based methods, other algorithms have demonstrated effectiveness in this area. In fact, malware imposes domain constraints that have encouraged researchers to apply other algorithms, notably reinforcement learning and genetic algorithms.

Note, methods for altering compiled binaries, e.g., Windows PE files, without damaging functionality are referred in computer science literature as static binary instrumentation, whereas the same methods are referred to as functionality preserving perturbations in data science literature. A number of synonymous terms—e.g., functionality preserving transformation/alteration and obfuscation/obfuscating transformation—makes codifying this vein of research an interdisciplinary challenge.

4.2.2 Domain Constraints of Binary Instrumentation

As alluded to in the previous sections, instrumenting a malware binary without damaging functionality is not a trivial task. Of the many file formats malware can exploit, the most challenging from an instrumentation perspective are the binary formats like Windows PE, Unix ELF, and Mac Mach-O. This is due to information loss in the compilation process. The most common target format in surveyed works has been Windows portable executable (PE), the native executable format for Windows. Both .exe and .dll files are examples of the PE format Bridge et al. (2021a). Although several papers do not provide the machine architecture of the PE binaries they utilized, the most common architectures recorded are X86 and its 64-bit version X86-64. X86 is a variable-length instruction set that may mix data with code, assuming the code section has the requisite permissions, in order to optimize runtime. Since opcode lengths in X86 are variable and code/data are mixed, differentiating a jump instruction from a data value that happens to correspond to a valid jump opcode is not trivial. Normally, analysts use reverse engineering tools like IDA Pro and Ghidra to recover data type and control flow information. This process depends heavily on heuristics, speculation, and analyst experience; it is not easily automated or scaled. Therefore, researchers have largely limited themselves to instrumentation methods that do not require full reverse engineering. Before discussing these methods, a short primer on the PE file format is provided.

4.2.3 Windows PE File Format

A PE file consists of a header with multiple sub-headers followed by code and data in discrete sections. The headers provide the program loader with requisite information for finding resources in the file and staging them in virtual memory so execution can begin. When discussing the structure of PE files it is important to consider the distinction between virtual address and file/raw offsets. The raw offsets correspond to the location of assets within the PE file as it sits on disk. The virtual addresses predict where these assets will sit in virtual memory. The raw offsets must conform to the *file alignment* field present within the (not so) *optional header*. The virtual address must conform to the *section alignment* field also found in the *optional header*. These alignment fields are necessary since memory is allocated in pages. Furthermore, memory protections are applied on a per-page basis, meaning that read-only data cannot be present on the same page as read-write data.

PE files begin with the *DOS header* which contains the signature or magic number used to identify PE files: the bytes 0x4D, 0x5A ("MZ" in ASCII) at offset zero. These bytes must be present at that offset or execution will terminate. Technically, these bytes mark the files as a DOS executable with a subsequent signature ("PE") at the start of the *NT header* actually identifying the file as a Windows executable. There are a number of optional fields following this DOS signature, culminating in the *e_lfanew* field. This field, always found at raw offset 0x3c, points past the *DOS stub* to the start of the *NT header* where the "PE" signature sits. All fields in the *DOS header*, with the important exceptions of the DOS signature and the *e_lfanew* field can be overwritten safely according to Demetrio Demetrio et al. (2021c).

The *DOS Stub* is a small DOS program that prints a string to the screen if a DOS operating system attempts to load the PE file. For this reason, it can be overwritten without damaging the PE file's execution on a Windows operating system. In fact, according to Microsoft's documentation Bridge et al. (2021a), the use of custom DOS stubs is not unexpected behavior.

The *rich header* is an optional header, which may reside between the end of the *DOS stub* and the start of the *COFF header* Poslušnỳ and Kálnai (2020). Ergo, the *e_lfanew* field from the *DOS header* points past the *DOS stub* and the *rich header*. This undocumented header may be added by Visual Studio's linker to track which Microsoft products were used in the creation of the PE file Poslušnỳ and Kálnai (2020). Poslušný and Kálnai found that, as of 2019, 73.20% of 1 million malicious PE files and 94.20% of 120,000 benign PE files contained a *rich header*.

The NT header consists of the PE signature ("PE\0\0"), the *COFF* (common object file format) *header*, and the *optional header*. The COFF header contains, among others, the relevant fields *number of sections*, *time/date stamp*, and *size of optional header*. The *size of optional header* is used to locate the start of the *section table* which follows the *optional header* Vuksan and Pericin (2011), while the *number of sections* field is used to iterate through the entries in said table. The *COFF* header ends with a *characteristics field* whose value determines whether the PE file is an EXE (can be run directly) or a DLL (can not be run directly); note, it is this field—not the file's extension—that determines whether the OS treats file as an EXE or a DLL. DLL files exist to export functions which can then be utilized by EXE files.

The next header, the *optional header*, is required for image files (.exe or .dll) but not for object files. The term arose from the extension of COFF files into the PE format. This header contains the aforementioned file and *section alignment* fields as well as the *address of entry point*,*size of image*, *size of (all) headers*, and *checksum* fields. The *size of headers* is aligned with the *file alignment* and is of note for some instrumentation actions. The *address of entry point* is the virtual address where the instruction pointer will be placed to begin execution once the PE file is loaded into memory.

Immediately following the *optional header* are the *data directories*. Each entry is a pair consisting of the relative virtual address and size of a particular resource or table, including the *import table* and *import address table* (IAT). The *import table* contains named entries for external functions imported from other files, namely DLLs. Initially,

the IAT contains the same entries, but is overwritten at the start of execution with the addresses rather than the names of imported functions. These tables are normally stored in an *.idata* section. The remaining entries are placed inside sections with corresponding permissions. Each entry is located by finding the section that contains the entry's virtual address.

Finally, we have the section table. Each entry contains a *name* (ignored by the program loader), *virtual size*, *virtual offset*, *raw size*, and *raw offset*, *characteristics field* and some additional optional fields. The characteristics field plays a critical role in specifying the read/write/execute (RWX) permissions that will be applied to the section once it is mapped into virtual memory. It is not uncommon to find a section with a larger *virtual size* than *raw size*; this indicates additional room is needed to store variables during executions. Packers such as UPX will create a section with a raw size of zero and a large virtual size that will, once the file is loaded into a process, receive the unpacked bytes from UPX's unpacking routine. Since *file alignment* and *section alignment* may differ, it is also possible to encounter sections whose *raw size* exceeds the *virtual size*. Ordinarily the last section table entry is followed by a number of padding bytes since the total size of the headers is padded according to the *file alignment*.

The code and data sections comprise the rest of the PE file, each with their own set of RWX permissions. Padding bytes are often present at the end of each section in order to match the *file alignment*. Additional sections are often added during compilation for debugging purposes. These additional sections can be stripped away, which is done to protect intellectual property or disguise the origin of malware. It is possible to add data not contained in a section to the end of a PE file without interfering with the file's execution or loading procedures using an *overlay append* action. This region is referred to as the overlay since it is not mapped into virtual memory.

Since the headers in a PE file are necessary to load the PE into memory, they are readily accessible without requiring reverse engineering. Open-source tools like LIEF

(https://github.com/lief-project/LIEF) and pefile (https://github.com/erocarrera/pefile) can parse these fields and provide some methods for altering them safely. The bulk of prior research manipulates these fields.

4.2.4 General Malware Detection

The detection of malicious activity is at the heart of operational cybersecurity. Maliciousness can be difficult to define, let alone detect. Towards that end, incoming files must be scanned to detect whether the files contain malicious capabilities. Keylogging, for instance, is a feature utilized by both commercial software as well as malware. In a benign program, the keylogging is used to provide some sort of service, like text prediction or auto-correction. In a malicious program, the keystrokes are used to capture login information. Consequently, the classification of malicious activity is not a trivial problem.

On a theoretical level, Cohen Cohen (1987) demonstrated that the assumption of a perfect malware detector is necessarily contradictory, i.e., no malware detector will ever achieve 100% accuracy. The question of malware detection, therefore, remains the domain of approximate solutions, wherein defenders maximize the performance of imperfect detection systems. To that end, malware detection systems, known informally as antivirus programs, use pattern matching, heuristics, and/or artificial intelligence to determine whether a file's capabilities are benign or malicious Aslan and Samet (2020); Bazrafshan et al. (2013).

Malware detection methods are either static or dynamic. Static methods treat the suspicious file as a string of bytes, attempting classification based solely on the file's static appearance in memory. These methods have played a key role in the development of adversarial methods for altering binaries as these are the features most readily applied (thanks to their high throughput) by defenders and altered (simplicity compared to altering dynamic behavior) by attackers. Dynamic methods execute the file in a sandbox (secure environment) and collect metrics based on actions

taken by the file. These methods can be evaded using anti-sandbox techniques that pose a problem when dynamic detection is used as an oracle for the true label of an adversarial sample targeted at a static detector. Both methods have advantages and disadvantages; some malware detection systems combine the two methodologies to develop a more comprehensive profile for analyzed files.

4.2.4.1 Dynamic Malware Detection

Dynamic malware detection executes suspicious files in a sandbox, which is a rigorously isolated virtual machine (VM) Aslan and Samet (2020); Or-Meir et al. (2019). Sandbox tools create an isolated environment where a guest operating system, complete with file structure, can be spooled up within a host operating system. Proper setup of a sandbox allows for the execution of suspicious files without compromising the host OS. By monitoring important areas of memory, like registry keys, a dynamic detection system builds a behavior-based representation of the potentially malicious file. The advantage of this methodology is that it can disregard any potential obfuscation applied to the suspicious file itself, which is otherwise a common malware-hiding technique.

Malicious authors routinely use a wide variety of methods to obfuscate the appearance of their code. These appearance obfuscations render simple analyses, like function extraction or string retrieval, useless unless additional work is applied first to unravel the obfuscations. For example, packing a file can fool signature-based detection Martignoni et al. (2007). Obfuscations can be so complex that manual reverse engineering by a well-trained and experienced cybersecurity professional is necessary to unravel the deception. However, this appearance obfuscation only applies to the file as it sits on disk.

Dynamic analysis ignores the static representation and, therefore, any static obfuscations that have been applied. Instead, dynamic detectors collect artifacts based on the malware's active behavior. In order for the file to complete its malicious objective, it *should* sooner or later reveal itself by attempting to read/write sensitive

files or Windows registry keys which are routinely monitored by anti-virus tools. However, malware authors have developed several behavior obfuscation techniques, also known as anti-sandbox techniques. These methods alter the file's behavior based on the presence of telltale artifacts associated with VM and/or debugging software, such as information on running processes, access patterns of the file system, or discrepancies in execution timing Bulazel and Yener (2017). If any artifacts are detected, the malware does not execute. Some malware simply uses a random function that prevents execution in some percentage of infections without checking for VM artifacts, thereby reducing the odds of a sandbox detecting it. This topic is revisited in Section 4.3.5.1 because of its significance to functionality preservation. In the end, dynamic analysis, like static analysis, consists of a cat-and-mouse game where defenders must constantly adapt to novel means of evading detection.

4.2.4.2 Static Malware Detection

Static detection examines a suspicious file without executing it Aslan and Samet (2020). Static methods can be used to detect malware directly on a host or network without first needing to route the file to a sandbox. Static detection is, in a general sense, faster than dynamic detection since there is no need to spool up a sandbox and wait an arbitrary amount of time for the malware to execute. However, static methods can be deceived by appearance obfuscations such as packing Muralidharan et al. (2022) or encryption. Even without obfuscation, static analysis is complicated by the compilation process that produces binary formats like Windows PE. When a file is compiled, it undergoes a transition from human-readable source code to machine code. This process destroys useful information contained in the source code. For instance, any objects in the source code are flattened into a series of arrays. Additionally, determining the data type of an argument–a trivial task in source code–requires a careful process of elimination that is usually performed manually in a process known as reverse engineering. This loss of information concerning the role played by each byte in the file limits functionality preserving changes to bytes whose roles can be readily

determined. It also means that some values for a byte may be valid, but others are not. Since some values are not valid, blindly altering bytes according to feature-space optimization may break functionality. Thus, researchers must limit their alterations to bytes whose role is known and non-essential (masking) or ensure that their system does not alter a funcitonal byte to an invalid value (bounding). These masks and bounds require knowledge of the PE file's structure and may require conservative disassembly.

Traditionally, static detection was achieved using signature matching Bazrafshan et al. (2013), wherein a cybersecurity operator identified a key set of bytes within a detected piece of malware. This signature was then propagated to all defended endpoints. Incoming files were then matched to a list of known signatures. This process introduced a delay (due to the detection/propagation cycle) and was only effective if the signature was not altered at all between infections. Polymorphic and metamorphic malware, however, are able to alter their own bytes with each fresh infection Aslan and Samet (2020). The rise of these two types of malware lead to the use of machine/deep learning rather than signatures. Once trained, machine/deep learning (ML/DL) models can detect novel malware provided this novel malware is similar to malware seen during training. This allows for the detection of polymorphic and metamorphic malware Aslan and Samet (2020). As with dynamic analysis, however, static analysis is a game wherein every new defensive technique inspires a new form of attack.

4.2.5 Experiment Design Terminology

This section provides the terminology routinely encountered in adversarial binary papers. Since this vein of research combines the disciplines of data science and computer science, domain-specific terms drawn from machine learning or cybersecurity are defined here for researchers from other domains. The most common metric for

reporting on adversarial binaries is the evasion rate or ER. It measures the percentage of files whose label flipped from malicious to benign after being altered.

4.2.5.1 Attack Scenarios

In a *white-box* attack scenario, the attacker has full access to the internal workings of a target modelmalware detector. Gradient-based methods are an example of a white-box attack since the attacker needs to recover the gradient of the loss function with respect to the input from the target model. This gradient tells the attacker which changes to the input will have the most significant effect on the label. Since a malware detector's output is binary (malicious or benign), we are not typically concerned with changing one label into a particular other label (i.e., targeted attack), as might be done in the image domain. For binary labels, it suffices merely to change away from the existing label (i.e., untargeted attack). The white-box scenario is not considered realistic in the malware detection domain, as it assumes full access to a malware detector. This is typically not available since vendors are highly motivated to keep the internal workings of their models secret to protect intellectual property and prevent malware authors from learning how to evade their detection tool.

A *grey-box* attack is more realistic, as it assumes only partial knowledge of a target model. This knowledge may take the form of the feature set (e.g., the EMBER feature set discussed in Section 4.3.1), the model architecture (e.g., neural network), or the actual numeric score applied by the model ($x \in \mathbb{R} | 0 \leq x \leq 1$) rather than the binary output (benign or malicious).

The *black-box* attack is the most realistic, as it assumes no knowledge of the attacker's model. Note, some authors describe a scenario where the numeric score output by the model is revealed rather than just the label applied by the model. They refer to this as a *soft-label* scenario. If only the final label is revealed, then the term *hard-label* is used. Since commercial malware detectors in practice rarely reveal scores, we argue the *soft-label* scenario is a form of grey-box scenario not a type of black-box scenario.

In a truly realistic scenario, the adversary would only be aware of success if their malicious program succeeded in establishing contact with the malware author via a reverse shell or some other covert communication channel. Since this realistic scenario would introduce an excessive time delay during training, the conventional approach is to use a surrogate as the classification model.

4.2.5.2 Surrogate Models

The surrogate is a malware classification model trained to detect malware and hosted by the attacker/researcher. By using a surrogate hosted on their own machines, researchers are able to train adversary models more quickly and avoid leaking information to a target model. The latter trait is especially important when the target model is actively monitored for novel threats, e.g., a cybersecurity endpoint detection and response (EDR) tool that learns from new data (ML-based) or can be updated with new signatures (signature-based).

Note, pairing an attacking model with a surrogate target model is not the same as defining a generative adversarial network (GAN). It is true that a GAN consists of a pair of models: a generator (attacker) and a discriminator (target). However, in a GAN both models are trained simultaneously so that the attacker can gradually improve Goodfellow et al. (2020). In the more general attacker and surrogate setup, the surrogate model may be fully trained at the outset, with only the attacker learning over time.

4.2.5.3 Transfer Property

A surrogate model can be used to convert the gradient-based white-box attack into a black-box one, since the surrogate approximates the necessary gradient from the target model. This approach leverages the experimentally established *transfer property* Szegedy et al. (2013); Papernot et al. (2016a,b), which dictates that an adversary model trained against a surrogate classifier will likely be effective against an unknown

target classifier, regardless of differences in the internal architecture. This property is attributed to the fact that the surrogate and target classifier have been mathematically optimized for the same classification task. Therefore, an attack optimized against model A will be reasonably effective against model B if both models are trained on the same task.

However, this property assumes some crossover in the feature space between model A and model B. There are published research models, covered in more detail in Section 4.3.1 whose extracted features do not overlap at all, meaning that there exist perturbations which change some of the features extracted by model A but *none* of the features extracted by model B. A perturbation which does not change any feature at all (as in model B's case) can not change the label applied by model B. Ergo, the transfer property can only apply when two feature spaces share at least some features, which is not a valid assumption in the malware classification domain.

4.2.6 Optimization Algorithms

4.2.6.1 Algorithm Selection Considerations

Although the earliest adversarial methods focused on gradient-based methods for altering images, the malware classification domain imposes certain restrictions that have motivated the use of other, less efficient optimizers like reinforcement learning and genetic algorithms. Anderson, in the first papers to use RL Anderson et al. (2017, 2018), made the following arguments in defence of the RL approach.

- First, an RL algorithm can learn from binary output (i.e., a black-box scenario) without the need to extract the loss gradient with respect to the input from the target or surrogate model.
- Second, RL algorithms are effective in black-box scenarios where nothing is known of the target model's feature set and architecture.

- Third, a functionality preservation oracle is not necessary since the actions selected by the Rl algorithms can be written to respect functionality.

The first argument could be countered via the use of a surrogate. However, an effective surrogate depends on having some knowledge (i.e., a grey-box scenario) of the target model's feature set in order to leverage the transferability property. Moreover, a reconstruction method must be developed to map from the feature space changes dictated by the gradient back to the input Pierazzi et al. (2020); Eykholt et al. (2018). This mapping crosses the embedding space, which is not always differentiable Demetrio et al. (2021c). Prior authors have developed reconstruction methods to address this issue (see Section 4.2.6.2), but reconstruction methods are dependent on the embedding space used by a target or surrogate model. Some models, like MalConv, feature a straightforward embedding of raw bytes in a feature space that can be readily inverted Kolosnjaji et al. (2018b). Therefore, these reconstruction methods have, thus far, been restricted to models that use raw bytes as features (see Section 4.2.6.2). For other targets, e.g., any model that uses the EMBER feature set (discussed further in Section 4.3.1), the mapping from input to feature is a many-to-one operation that loses information Anderson and Roth (2018a). Ergo, mapping backward would require simplifying assumptions to be applied based on how individual features are calculated. Continuing the EMBER example, the raw sizes of sections are binned into a vector of length 50 using the hashing trick (intentional hashing collisions), meaning that the names of the sections which hashed to that index are lost along with the raw size of each section that contributed to the signed value stored in that index. To generate a mapping from this feature back to input, a list of section names that hash to each index would need to be populated and used to determine which sections of a given file mapped to each index. The change in the hashed raw size dictated by the gradient would then need to be propagated to on or more of the matching sections. Since RL does not need to map from the gradient (which exists in feature space) back to the

input, this reconstruction step is not necessary. This simplifies the development of novel actions.

With regard to the second argument, an examination of the Anderson action set (discussed in detail in Section 4.2.2) shows that all but three of the actions are targeted at structural features. Only the *upx pack/unpack* would affect the features extracted by a behavior-based model since the unpacking routine would alter the control flow. The *new entry point* provides a very small change to the behavior of the file by forcing the PE to jump from the new entry point back to the original entry point before execution continues as before. Ergo, some assumptions about the target model's feature set are implied by the Anderson action set's focus on static features. A truly feature agnostic method would include a suite of actions targeted at both static and behavioral feature sets. This is similar in concept to Rosenberg's approach Rosenberg et al. (2018) where a hybrid model was evaded using one static action and one behavioral action. To validate Anderson's assertion, the RL agent would also need to be trained on multiple target models (at least one based on static features and one based on behavioral features) or a hybrid model so that it could learn to adapt which type of action it selects (static or behavioral) over time. The second argument, therefore, needs to be validated using a similar approach to Rosenberg and, perhaps, the development of behavioral perturbations on a larger scale than adding an unpacking routine or new entry point.

Song Song et al. (2021) and Labaca-Castro Castro et al. (2019b) later demonstrated that the third argument is invalid since the LIEF tool used by Anderson is known to break some binaries during reconstruction. Therefore, an oracle that validates functionality of altered samples is necessary in order to prove that a loss of functionality—which could negate the true label of the sample (discussed in 4.2.1)—has not occurred.

In light of these challenges, we argue that RL has not yet demonstrated a superior approach. Both RL and gradient+reconstruction methods can generate adversarial binaries. Head-to-head comparisons across algorithm families are rare in published

literature and highly limited in scope. What does emerge from our survey, though, is the importance of input-space alterations regardless of how they are guided. First, assumptions as to functionality preservation are not safe (see Section 4.4.1). By producing actual binaries, researchers can verify functionality via an oracle like Cuckoo, a well-known sandbox (see Section 4.2.4.1). Second, malware detectors may be trained on a wide variety of structural or behavioral features. Datasets composed of adversarial binaries enable feature and model-agnostic adversarial training. By training existing detectors on properly labeled adversarial samples, malware detection vendors can inoculate their models to known perturbation methods Anderson et al. (2018); Wu et al. (2018); Fang et al. (2020). If the perturbations were applied in the feature space, generated adversarial features could only be used for adversarial training of models with identical feature sets. Since malware detection vendors are highly motivated to keep their features and architecture secret, this would limit how widely adversarial samples could be shared across industry. In order to standardize defense against these samples, adversarial data sets should be agnostic to feature-extraction routines, i.e., actual binary files. That way, each vendor can extract the necessary features unique to their model from public datasets without leaking information, allowing them to determine which alteration methods pose a threat to their particular feature set. There are also attacks on feature extraction itself that it might be useful to model rather than assuming the extracted features are fully representative of the input sample Koch et al. (2022); Jana and Shmatikov (2012b).

4.2.6.2 Gradient-based Algorithms

This family of methods exploits the differentiable nature of ML models which utilize backprogation in order to guide the perturbation process. Although gradient methods were first demonstrated in the image-classification domain Goodfellow et al. (2014b); Szegedy et al. (2013), they have since been adapted to the binary classification domain. The methods can be either targeted or untargeted Kurakin et al. (2018). In both cases, the gradient of a loss function with respect to the last differentiable layer (usually the

embedding layer for a neural network-based malware classifier) is calculated in order to reveal which feature space changes will most affect the loss function.

The untargeted version seeks to maximize the loss function which is used to score the model's performance across all classes. Increasing the value reported by the loss function equates to causing the target/surrogate model to misclassify a sample. Since this attack is not targeted at a particular class, it could be used to both make malicious sample seem less malicious and benign files seem more malicious.

Since the false negative case is generally the most concerning for security researchers, most authors use a targeted formulation that assumes the input has a true label of $y = 1$ (malicious binaries). In this case, the objective is to minimize a loss function which measures the distance between the model's prediction ($y \in \mathbb{R} | 0 \leq y \leq 1$) and the desired label ($y = 0$).

Since feature extraction is not always a differentiable process, a mapping from the optimized feature representation back to the input must be calculated. This step is often referred to as input-space reconstruction. For a model trained on raw bytes like MalConv, reconstruction is straightforward even though the embedding is not differentiable. MalConv embeds each of the 256 possible byte values in a binary to a higher dimension embedding space Raff et al. (2018b). Kolosnjaji Kolosnjaji et al. (2018b) and Demetrio Demetrio et al. (2021c) used the *closest positive* method to reconstruct a binary using gradient optimization. The process is summarized below and in Figure 4.1.

- The gradient of the loss function with respect to the input is calculated
- A line is generated in the embedding space whose origin is the current embedding value of a given byte and whose direction is specified by the gradient
- The new embedding value is determined by finding the closest embedding value that is aligned with the direction of the gradient.

- Since the embedding values have a one-to-one mapping with the input bytes, conversion back to an input byte is then trivial.

Instead of using the full value of the gradient to update the feature representation, some authors (Kreuk Kreuk et al. (2018), Chen Chen et al. (2019), Suciu Suciu et al. (2019)) use the fast gradient sign method (FGSM) Goodfellow et al. (2014b) which uses only the sign of the gradient to update the existing feature space representation. In this case, the input reconstruction method simply maps the updated features to the nearest embedding value, which is referred to as the *closest* reconstruction method Demetrio et al. (2021c).

For other models, such as EMBER, the feature extraction does not have a one-to-one mapping, so simplifying assumptions would need to be made. EMBER also uses the hashing trick (intentional hashing collisions to bin variable length input into a fixed number of bins) on section table information Anderson and Roth (2018a). In order to map back to the section(s) which generated the values in each bin, a mapping from input section names to their respective bins would need to be generated and any changes to said feature mapped to one or more of the sections that contributed to that bin. This mapping, while not trivial, is tractable. However, the problem of functionality preservation arises. Suppose the optimal perturbation would decrease the virtual size of each section or make every section non-executable. The resulting binary would not function. Ergo, the feature changes would need to be bound to values that would not damage functionality, which would be different for each binary. While authors in the surveyed papers have used masks to prevent any alteration to key bytes, we have not seen upper/lower bounds applied on a per-feature basis in existing publications.

A generative adversarial network or GAN Goodfellow et al. (2014a) is a pairing of models where the first model generates a sample and the second model classifies the sample as either authentic or machine-generated. This approach has been adapted to the binary instrumentation task by adjusting the classifier to apply a

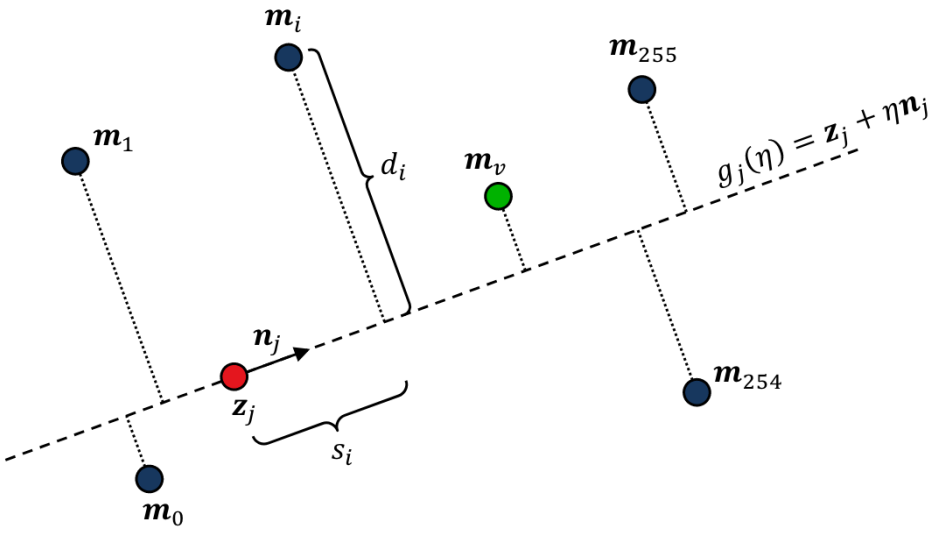


Figure 4.1: Input-space reconstruction algorithm used by Kolosnjaj Kolosnjaji et al. (2018b) and Demetrio Demetrio et al. (2021c) to map gradient-based optimization back through the non-differentiable embedding to the input binary file. z_j (red) is the original feature space embedding. n_j is the normalized gradient. m_v is the nearest embedding in the direction of n_j, so it becomes the new embedded value. Since the embedding is 1-to-1 with the input, m_v is mapped back to a byte and written to the (now) adversarial binary Demetrio et al. (2021c).

maliciousness score rather than a real/fake label. This score is then used as feedback. A GAN is distinct from an attacker+surrogate pairing since both models in a GAN train simultaneously to allow the generator enough initial success to avoid an early performance plateau. If the discriminator is "too good" too soon, the classifier can fail to improve Goodfellow et al. (2014a). The single surveyed paper that used a GAN did not attempt to generate an entire binary using the generator. Instead, the generator created a vector that was used to select combinations and variations of the discrete actions the authors had developed Chen et al. (2019). Directly generating a functional binary using a GAN would require extremely high fidelity output from the generator since a single incorrect byte would invalidate the generated file. Instead, the authors created a set of functionality preserving actions and merely relied on the GAN to generate an instruction vector.

4.2.6.3 Genetic Algorithms

Genetic algorithms (GAs) are a biologically inspired method for optimizing samples in a population Holland (1992). To deploy a genetic algorithm on a problem, the features of interest first need to be mapped to a genetic representation that is typically a binary string. It is therefore convenient that file-based malware, like all files on a computer, are stored as a binary string. Note, input length is an issue that must be considered since malware files can be several gigabytes in size. In this case, a truncation or compression mechanism would be needed if attempting to map the raw bytes of a file to a fixed-length genetic representation. An alternative would be to first extract a fixed length discrete representation (e.g., the EMBER features) before converting to a binary string.

Next, the *fitness* function, *crossover mechanism*, and *selection criteria* must all be defined. Each sample is scored according to the *fitness* function. High-scoring samples are then *selected* for *crossover*. Crossover blends the traits of a pair of selected samples in order to produce a new sample with a higher fitness score. Since this approach could lead to a local optimum rather than a global one, *mutation* is used to randomly

alter traits so that successive generations do not converge to a single, identical set of traits that can no longer meaningfully *crossover*.

In our context, a fitness function can be used to score how effective an action is at altering a binaries classification, minimize the scale of the perturbation, penalize non-functional perturbations, increase/decrease diversity using a distance metric, or any combination thereof.

4.2.6.4 Reinforcement Learning

Reinforcement learning experimentally determines the optimal or near-optimal course of actions to maximize cumulative reward in a game-like scenario Sutton et al. (2018). RL is useful in situations where the benefit or detriment of a particular action can not be determined until the conclusion of the game several steps later. RL is a unique third branch of machine learning since it does not fall under the umbrella of either supervised or unsupervised learning.

Supervised learning requires a training set that consists of input data mapped to correct output labels. RL is distinct from supervised learning since *a priori* labels for the value of an action at a state are not available Sutton et al. (2018). These values must be determined experimentally by playing several iterations of the scenario to completion.

Unsupervised learning does not require output labels in the training data. Instead, input samples are grouped according to similarity using a variety of algorithms. RL is distinct from unsupervised learning since RL seeks to optimize an action rather than uncover structures within the data Sutton et al. (2018).

The reinforcement learning problem is formulated as a Markov decision process (MDP) that maps states to optimal actions using reward signals in order to maximize cumulative reward. The mapping generated by training an RL algorithm is known as the policy. Q-learning, a popular example of off-policy temporal-distance learning, is

defined thusly Sutton et al. (2018):

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{4.1}$$

- $Q(S_t, A_t)$ is the action-value function that estimates the value of taking action A_t in state S_t
- $a \in A$ is the action chosen from the set of possible actions A
- α is the step size
- γ is the decay rate which encourages the model to take profitable actions sooner rather than later
- R_{t+1} is the estimate of the expected reward at time $t + 1$

Although there are many different RL algorithms, they all use the same basic elements of state, action, and reward to generate a policy. In tabular RL, a table tracks the value of taking each possible action from each state. During training, the values in the table are updated. During evaluation, the action with the maximum cumulative reward (as recorded in the table) is chosen in each state. This approach is computationally expensive for games with a large number of possible states. Therefore, deep reinforcement learning utilizes a neural network in place of the table. This function approximation via a deep neural network has the added benefit of allowing the model to generalize, provided similar reward distributions, to the values of actions in states that have not been reached during training Sutton et al. (2018); Mnih et al. (2013).

4.3 Surveyed Methods

4.3.1 Target Models and Feature Sets

In this survey, we found that the the target(s) model is not consistent across papers, although MalConv and gradient boosted decision tree (GBDT) appear most often. Commercial off-the-shelf (COTS) anti-virus (AV) models with unknown feature sets and architectures have also been tested. MalConv is trained on raw bytes while the GBDT is trained on EMBER features Anderson and Roth (2018a). When discussing adversarial samples the feature space is a significant factor, perhaps more so than the model architecture thanks to the transferability property Papernot et al. (2016a). Although actions tend to transfer between models trained on the same task, this transfer is not possible if a model omits the feature(s) altered by an action. We therefore move on to an examination of feature sets used by prior researchers.

EMBER is a set of features specific to the Windows PE file format. It includes header fields, section table entries, and some summary statistics like the byte histogram and byte-entropy histogram. The feature set could be utilized by model architectures other than GBDT like neural networks or support vector machines. The EMBER feature set is primarily structural in nature as opposed to behavior/functional or language-based (Ngrams) since the features are drawn from the header fields that specify how to load the PE file into memory prior to execution. The features do not include, aside from the byte histogram and byte entropy histogram, opcodes or variables used by the malware during execution.

Another feature set are the raw bytes themselves as used by the MalConv model introduced by Raff et al. Raff et al. (2018b) and the CNN proposed by Coull Coull and Gardner (2019). Since these models are trained on raw bytes, they are file-format agnostic in theory. In practice, they are trained specifically on Windows PE files. MalConv utilizes a deep learning architecture with 1D convolutional layers and a gating mechanism to reduce the computational load of training on the raw bytes.

Since PE files can be several gigabytes in size, models trained on raw bytes can be evaded by appending malicious content to the end of a binary whose size exceeds the model's maximum input length. Therefore, Raff et al. followed up with MalConv2 which increased the maximum file length consumable by the model Raff et al. (2020a). The truncation vulnerability could also be reduced by head and tail scanning rather than reading in the first N bytes. One important attribute of the MalConv model is translation invariance thanks to the global max pooling layer. Since the location of a significant byte chunk is lost during analysis, adversarial data appended to the overlay is just as effective at altering the label as data inserted in the header, even if this overlay data is never referenced during loading or execution. Therefore, the expert knowledge needed to alter a binary so it can evade MalConv yet remain functional is much lower than for a model like EMBER incorporates several fields from the headers and section table. These fields must remain valid when altered so the program loader can accurately load the file into virtual memory. Coull's CNN uses a hierarchical series of convolutional and max pooling layers in order to relate low-level local features with one another before they flow to higher level features Demetrio et al. (2021c); Coull and Gardner (2019). As with MalConv, the pooling layer is followed by a fully connected layer.

A third feature set is not based on the bytes (raw or parsed) but on the control flow graph (CFG). This graph tracks the instruction pointer as it jumps from one basic block to another. This feature set is typically used with a graph neural network or convolutional graph neural network architecture where the model learns to label the graph generated by a malware program Zhang et al. (2022). The CFG must first be extracted by a reverse engineering tool like IDA Pro or Ghidra, slowing the feature extraction process. It's worth noting that actions which alter the structure but not the control flow of a PE file would affect EMBER features but not the CFG, meaning that a CFG-based model is largely immune to the Anderson action set (except for the upx packing/unpacking actions). We believe this is a strong imputus for industry partners to replace raw or structural features with behavioral features like the CFG.

This would reduce the attack surface considerably, although it should be noted that even these features can be spoofed.

Why have most prior researchers focused on structural rather than CFG features? Structural changes are much simpler to enact. In order for a PE file to be parsed, the structure of the headers must conform to the PE file format specifications, providing context and purpose to many of the bytes in the headers. In the X86/X86-64 processor architecture, for instance, code and data beyond the headers are intermingled and overlapped, making reverse engineering difficult and alteration highly prone to destructive interference. By instead focusing on features generic to all PE files, prior researchers are able to affect features without first needing to reverse engineer the contents of the code/data sections. Additionally, the header is agnostic to processor architecture, so a wider variety of files are alterable by the same action set.

Zhang Zhang et al. (2022) avoided the difficulty of altering code/data without loss of functionality by making their alterations in the feature space not in the input space (the actual file). Although feature space alterations are useful for adversarial training, it is difficult to argue that they represent a real-world threat. Additionally, feature space alterations can only be used for training of classification models with the same exact feature space. Altered malware binaries, however, can be used for adversarial training of any malware classifier (assuming file format coverage).

There are tools which can make alterations to code/data like those used by Zhang on actual malware instead of the feature set extracted from said malware, but they require more expertise than the structural actions popularized by Anderson. These tools include but are not limited to Angr, E9 Patch, and the Grammatec suite of tools (DDisasm, GTIRB, and GTIRB pretty printer) Schulte et al. (2019). The Grammatec tools are particularly promising as they utilize an intermediate representation (GTIRB). This intermediate representation is a higher level abstraction of an executable file, meaning that changes made to the intermediate representation could be mapped back to multiple executable file formats not just PE. We are not aware of any academic publications that lift malware files to an intermediate representation before altering

them in order to evade classification by an AI model. For further reading on the subject, there is a comprehensive survey of methods and tools by Wenzl Wenzl et al. (2019). More recently, Shulte, Brown, and Fultz evaluated 10 binary instrumentation tools on a large corpus of binaries, comparing the functionality preservation, runtime increase, and file size increase incurred by altering these binaries Schulte et al. (2022a).

Given the distinct nature of these feature sets, it is difficult to conclude that any optimization algorithm is best across the board. Instead, we can only provide conclusions when the action set is held constant.

4.3.2 Gradient, Genetic, and Other Guidance Mechanism Papers

This section is summarized in Tables 4.4 and 4.5. Rosenberg initiated the gradient-based method for altering malicious binaries by developing a pair of methods targeting two different feature sets (static strings and dynamic API call sequences) and multiple variations of nine different classification models (DNN, CNN, RF, ,SVM, GBDT, LR, RNN, GRU, LSTM) Rosenberg et al. (2018). A surrogate model (GRU) was used so that the attack did not rely on internal access to target models, demonstrating that attacks crafted using the surrogate transferred well to other architectures that use the same features. Two sets of experiments were conducted. The first focused on target models and a surrogate model trained solely on a dynamic feature set, which consisted of a 1-hot encoding of the first 140 API calls made by a malware fileas reported by Cuckoo, the dynamic analysis tool. RNN, GRU, and CNN models were the most vulnerable (100% evasion rate) while LR and SVM were the least vulnerable (70% and 71%, respectively) to *FGSM API hooking*. One obvious limitation of this experiment is that real-world malware detectors can draw from a host of features. Therefore, Rosenberg created a second experiment.

This time, the target model (an LSTM) was trained on the API call sequence and the most common strings. Three attacks were launched using a GRU surrogate. The

first used only the API hooking action, achieving an ER of 46% compared to an ER of 96% on the dynamic-only target model. The second only uses the string append action, resulting in an ER of 69% compared to an ER of 77% against a static-only model. The final experiment used both actions (a hybrid attack) against the hybrid model with an ER of 82%, indicating that mixing feature sets blunted the attack somewhat.

Lastly, Rosenberg evaluated the functionality preservation of their API hooking action, finding that all of the malware samples tested still executed the same sequence of original API calls along with the newly inserted ones after alteration, indicating that the additional API calls did not damage the malware's ability to function.

This paper set a high bar by demonstrating the transferability of attacks crafted using a surrogate, creating a tractable reconstruction method for mapping the feature gradient back to the binary, and evaluating functionality preservation using a precise metric. Rosenberg made one other contribution of note by filtering malware prior to the experiments: any malware that did not make 15 or more detectable API calls when tested in the sandbox was rejected. The goal here was to discard any files that might be using anti-sandbox methods to conceal their functionality. This step allows for confidence that all malware actually executes detectable behavior prior to alteration. This not only reveals that the malware was indeed functional and detectable prior to alteration but also increases our confidence that any loss of functionality is due to the alteration not anti-sandbox behavior. Note, if the anti-sandbox method in question is simply randomized, a malware file might appear to be functional prior to alteration but appear nonfunctional after alteration. Repeated execution of the file before and after alteration could be used to mitigate this potential issue.

Kolosnjaji demonstrated the first gradient-based method for directly altering bytes which are then appended to the overlay of a PE file. Since data appended to the end of a Windows PE file is not mapped into the virtual memory of a process unless a corresponding and valid section table entry is added to the section table in the headers, this is widely regarded as a safe location for adding content to a PE file. Initially, a

vector of random bytes is appended to the overlay. Once these bytes are embedded into features, these feature values are updated using the full gradient, not just the sign of the gradient. A surrogate or white-box target trained on raw bytes is needed in order to map from the target's label back through the feature extraction process to the raw byte values in the overlay. Since the feature vector optimised by the gradient will consist of real values in the embedding space, a reconstruction algorithm is needed to convert back to raw bytes (integers). Kolosnjaji used the *closest positive* algorithm reviewed in Section 4.2.6.2.

Kolosnjaji's attack scenario is white-box since it attacks MalConv using gradient values extracted from MalConv. Functionality verification is not performed. One constraint of this approach is that models trained on raw bytes can only ingest so many bytes. If a malware file meets or exceeds the model's maximum capacity, the remaining bytes are merely truncated. This means that an append attack can only be successful if the length of the binary and appended bytes is less than the max capacity of the model. The authors compared their gradient-optimized bytes with random bytes, finding that appending 10,000 bytes achieved an ER of $<$20% using random bytes and 60% using gradient optimization, indicating that the gradient optimization was a significant factor for evasion. The target model was MalConv, which was trained on benign files scraped by Kolosnjaji and malicious files pulled from VirusShare and other sources.

Kreuk used iterative FGSM to optimize bytes added in two locations: the overlay (as with Kolosnjaji) and section padding areas. As mentioned in 4.2.3, the contents of a PE are laid out in sections which are page aligned, which means that padding bytes (null values) are added until the section contents are a multiple of the alignment. The size of the padding area can be calculated by subtracting the virtual size from the raw size. In effect, this subtracts the useful bytes that will be written into process memory from the padding bytes which will not be written into a process. Note, it is possible for the virtual size to exceed the raw size in some cases since the page sizes differ Ronen et al. (2018). UPX-packed files typically contain one section with a large virtual size

and zero raw size, indicating that—once the file is loaded into a process, bytes will be unpacked and placed in the pages allocated by the section. Kreuk determines which bytes are padding only, then uses FGSM to optimize their value. The target model was MalConv, which was trained on benign files collected by Kreuk and malicious files released by Microsoft Ronen et al. (2018). Using both of the *FGSM Overlay Append* and *FGSM Section Append* actions, Kreuk reached an evasion rate of 99% against MalConv using less than 1000 modified bytes. Functionality preservation was tested, but no details are provided on the tool used; the behavior graphs are simply reported as identical before and after alteration. The reconstruction method used to map the optimized features back to raw bytes was *closest*, as described in Section 4.2.6.2.

Demetrio published a trio of papers on this topic Demetrio et al. (2019b, 2021a,c). The first used an explainability technique, integrated gradients Sundararajan et al. (2017), to argue that MalConv places significance on the bytes in the DOS header and DOS stub, which are not vital to functionality Demetrio et al. (2019b). Demetrio's *partial DOS* Demetrio et al. (2019b), *full DOS* Demetrio et al. (2021c), and *extend DOS* Demetrio et al. (2021c) actions target this section. *Partial DOS* overwrote the DOS stub only, while *full DOS* overwrote the DOS stub and DOS header. Note, the DOS signature ('MZ') and e_lfanew pointers were not overwritten to preserve functionality. *shift DOS* increased the number of bytes available in this region for manipulation by shifting the optional header and all sections to a higher offset. Demetrio's other gradient-based action is *extend section*, which creates room to insert a payload of bytes after the headers but before the first section. This is accomplished by increasing the raw offset of each section table entry and then moving the corresponding section contents.Demetrio et al. (2021c). All of these methods used the full gradient and *closest positive* reconstruction method 4.2.6.2. Demetrio's second paper took an alternate approach using a genetic algorithm to train in a more realistic black-box, testing both soft and hard labels (see definitions in 4.2.5.1) Demetrio et al. (2021a). Two actions were paired with the genetic algorithm: *benign overlay append* and *benign section append*. The former pulled byte chunks from benign files and appended them

to the overlay. The latter action overwrite the padding bytes in existing sections with chunks from benign files. The genetic algorithm sought to minimize two terms: the maliciousness score/label applied by the target model and the size of the payload written to the file. Demetrio found that using a larger penalty for the size of the payload produced higher evasion rates than a smaller size penalty. With parameters held constant (number of queries and penalty coefficient), the evasion rate for *benign overlay append* only dropped 1-3% when switching from soft to hard labels, while the evasion rate for *benign section append* dropped 1-6%. In terms of target models, MalConv was more vulnerable to *benign overlay append* (up to 63%) with soft labels while a GBDT trained on EMBER features was more vulnerable to *benign section append* (up to 57%) with soft labels). With hard labels, *benign overlay append* was more effective than *benign section append* on both models. Demetrio also tested *benign section append* against 9 commercial tools on Virustotal, finding that their *benign section append* action transferred well to novel detectors, evading detection at a higher rate (11% to 64%) than merely appending random data (.5% to 39%). The GBDT and MalConv were both trained on the EMBER data set Anderson and Roth (2018a). Demetrio's third paper consolidated prior work in a head-to-head analysis of white-box and black-box scenarios Demetrio et al. (2021c). Demetrio's gradient-based and genetic approaches were both tested along with Kolosnjaji and Kreuk's actions against several convolutional neural network variants (MalConv, Coull's CNN). Note, Demetrio refers to Coull's model variants as DNNs, which is valid yet confusing considering that Coull refers to the models as CNNs. Two variants of Coull's model were trained on proprietary datasets while the rest of the models were trained on binaries from the EMBER dataset Anderson and Roth (2018a). The white-box scenario excluded the EMBER GBDT since it is not differentiable. The remaining models were all susceptible to all of the gradient-based attacks, with MalConv in particular suffering a high evasion rate from all actions except *shift section*. Coull's CNN, trained on EMBER binaries, outperformed the other models on all but the *extend* and *shift* actions. Demetrio attributes this to the fact that Coull's model incorporates some

sense of location via hierarchical convolutions whereas MalConv does not. A test of transferability also revealed that all of the models trained on raw bytes were vulnerable to transfer attacks (attack trained on model A, but tested on model B) while the EMBER GBDT was not. Moving on to the the black-box scenario, we see more interesting results. The evasion rate for all actions against the EMBER GBDT was marginal, except for the *genetic benign section append* action (ER $>$ 70%), which was also strongly effective at evading MalConv (ER $>$ 90%). The Coull DNN trained on EMBER binaries performed well against *gradient partial DOS*, *gradient overlay append*, and *genetic benign section append*, but failed significantly against *gradient full DOS* (ER of 20%), *gradient extend DOS* (ER $>$ 40%), and *gradient shift sections* (ER $>$ 90%). Demetrio argues that the effectiveness of the genetic method is due to the minimal representation of raw bytes in the EMBER feature set and the fact that drawing content from known benign files sidesteps the need for effective gradient updates, which are not available from an EMBER GBDT Demetrio et al. (2021c). As discussed in 4.3.1 EMBER features only represent the byte values in sections or the overlay via the byte histogram and byte-entropy histogram features.

Chen developed a number of methods for selecting which bytes to append to the overlay and compared their performance against MalConv Chen et al. (2019), finding that that their *FGSM+GradCAM benign overlay append* action outperformed all others, including *FGSM overlay append*, at all payload sizes (ER of 99% at max payload of 20000 bytes). Chen's first action, *random benign overlay append* randomly selects chunks from benign files and appends them to the overlay. The second action, *experience benign overlay append*, examines successful trajectories (sequences of actions that evaded detection) from the random selection action and appends the most successful (frequency of appearance in successful trajectories) chunks to append. The third action, *GradCAM benign overlay append*, used the GradCAM Selvaraju et al. (2019) explainability method to find chunks in benign files that are considered highly significant by the model. The final method, *FGSM GradCAM benign overlay append*, took the chunks selected via GradCAM and optimized them using FGSM.

This approach was not only the opt scorer for all payload sizes but also showed a massive improvement over the other methods with the smallest payload, achieving an ER of 74% compared with 1-9% for all the others. Chen argued that the significant gain is due to the sparsity of the gradient vector thanks to the max pooling operation used by MalConv Chen et al. (2019). By starting with known significant chunks, the gradient updates were much more effective than they were when starting from random bytes that are unlikely to, initially, have a large gradient associated with them. Chen made another interesting contribution by noting that adversarial training, a common defense based on fine-tuning a model using adversarial samples Goodfellow et al. (2014b), may lead to data poisoning (misclassification of benign files) if too large a percentage (1.4% for Chen's dataset) of adversarial binaries are added to the training data; instead, Chen proposed filtering out adversarial samples to prevent data poisoning of malware detectors that update regularly using freshly detected malware. Their approach, detecting when a PE's executable section hashed to the same value as the exec section from a different PE file, worked perfectly on data appended to the overlay but would not be effective against an attacker who altered any bytes in the exec section or used a fresh executable section.

As with Chen, Suciu compared append strategies, using methods drawn from Kreuk (single-step versions instead of iterative) and Kolosnjaji as well as their own *benign overlay append* method and a baseline *random overlay append*. The novel action, *benign overlay append*, draws chunks from the beginning of benign files under the assumption that these header bytes are more significant Suciu et al. (2019). They tested three version of MalConv that differed only in the dataset: binaries from the EMBER dataset (1.1 million), binaries pulled from Virustotal and other sources (16 million), and a mini dataset (8,598) drawn from the largest dataset. Suciu found that FGSM was the most effective append strategy against all versions of MalConv. Against intuition, the version trained on the largest dataset was the most vulnerable, which Suciu blames on the larger model encoding more information and therefore being more vulnerable to a single gradient update Suciu et al. (2019).

Suciu then compared *FGSM overlay append* to *FGSM section append*, finding that, although adding bytes to existing sections outperforms for some small payload sizes, *FGSM overlay append* can yield a much higher evasion rate (15% vs 33% against MalConv trained on EMBER binaries) because it can scale to larger payloads. The payload size for *FGSM section append* has, as implemented here, an upper bound imposed by the sum of the padding spaces after each section. Note, this limitation is not fundamental; Demetrio demonstrated that new space can be created by updating the raw offsets in the *section table*.

Finally, Suciu found that only 2/400 attacks transferred from one version of MalConv to another, indicating that a single-step gradient update did not yield samples that transferred well.

Labaca-Castro followed a genetic algorithm approach Castro et al. (2019a) and compared it to their prior work Castro et al. (2019b), a system that randomly selected from a set of actions to find action sequences that evade detection. They would later develop a reinforcement learning-based version of this system (discussed in Section 4.3.4) Labaca-Castro et al. (2021), but did not compare the genetic and reinforcement learning systems to one another. The action set used by all of their systems is the same: the Anderson action set (see Section 4.3.5) minus *imports append* and *upx unpack*. These two actions were left out as Labaca-Castro found they routinely damaged functionality; Song Song et al. (2021, 2020) also found a high likelihood of damage for files altered by Anderson actions. In all three papers, the framework of Labaca-Castro's system is the same: malware is ingested, perturbed (action applied), tested for functionality, and then scored by a target model. If the modified binary is mislabeled by the target, it is stored. Otherwise, it repeats the process until it is either mislabeled or the maximum number of perturbations is reached. The random system produced functional malware 18% of the time, meaning that many samples had to be discarded. The functional samples were able to evade detection by Virustotal up to 80% of the time. Labaca-Castro therefore focused on functionality preservation for the genetic agent in order to reduce time wasted generating non-functional samples.

They capped the action budget at 20 generations, finding that exceeding that amount was highly likely to break functionality. The fitness function used to score generated malware was more complex than others, incorporating the score applied by a target model, the functionality as reported by a sandbox, distance from the original sample (to encourage diversity), and a term to prioritize newer generations. Labaca-Castro found that the genetic system learned to create a functional adversarial samples over the course of several generations, creating 450 non-functional samples at generation 20 compared to 2050 for the random agent. The authors did not compare the evasion rates of the random and genetic agents, but did note that their genetic agent scored similar evasion rates to Anderson's RL agent. They also found that attacks transferred well between commercial and research (a GBDT assumed to be trained n EMBER features) detectors with a cross-evasion rate from 18% to 82%.

Sharif's work is one of the most significant in this space because their action set alters both the static and behavioral features of the malware. Both of Sharif's actions alter the executable code within a PE file by conservatively disassembling the file and finding sets of instructions that can be moved or replaced without loss of function Sharif et al. (2019). As mentioned in 4.2.2, fully dissassembling a file is an unsolved problem that relies on speculation and experience. However, there are chunks of executable code than can be dissassembled with high confidence. Sharif limits the editable bytes to these regions. The approach is based on Pappas Pappas et al. (2012), who built a tool for in-place binary rewriting in order to prevent ROP (return oriented programming) gadget extraction. Sharif's implementation of the tool allows for repeated instrumentation, which is necessary to iteratively modify a binary so it evades detection. The two actions implemented, *code displacement* and *code randomization*, are presented at length in Section 4.2.2. Since the bytes they alter are used during execution, these actions can alter static (bytes, byte histogram, byte-entropy histogram) and dynamic (basic blocks, register access patterns) features. The defense against adversarial samples proposed by Chen Chen et al. (2019) would not prevent Sharif's attack since the actions change the hash of the executable section.

Sharif's optimization algorithm begins with random actions. If a random action reduces the model's score (grey-box version) or moves the feature vector in the same direction (calculated using cosine similarity) as the gradient (white-box version), the action is kept. Note, the authors tested but did not report on a genetic algorithm variant due to poor performance. Sharif tested their black-box version against three neural networks: MalConv Raff et al. (2018b), Endgame Anderson and Roth (2018a) (a deep neural network trained on EMBER features), and AvastNet Krcal et al. (2018) (CNN trained on raw bytes). Malware files were drawn from the Microsoft competition Ronen et al. (2018) and benign files were pulled from public sources (package managers).

The authors first established a baseline by randomly perturbing a subset of malware files, finding that—sans a mechanism for selecting only the most effective transformations—the two actions only allowed 1/200 files to evade detection by any model. Next, they tested their gradient-based white-box methods, finding that *code replacement* was only effective against EndGame with an ER of 62%. *Code displacement*, on the other hand, evaded all three detectors 100% of the time. Sharif compared this to Kreuk's prior work, which also reached 100% evasion rate against AvsetNet, but narrowly underperformed on Endgame (99% ER) and MalConv (98% ER). Although Kreuk's simpler action performed very well, Sharif points out that non-functional appended bytes are more easily detected and filtered than displaced/randomized instructions.

In a black-box setting, AvastNet could not be evaded using Sharif's system, while MalConv and Endgame were evaded in 33% and 97% of test cases. Transferability was also tested against malware detection engines hosted on Virustotal (`virustotal.com`), where the average detection rate dropped from 55 for original binaries to 42 for randomly altered binaries to 33 for binaries altered by the most effective adversarial action.

Sharif tested 11 benign binaries—including a stress testing binary—for functionality preservation by directly executing them. The authors also tested 10 malicious binaries by running them in Cuckoo; all 10 malicious files "accessed the same hosts, IP

addresses, files, APIs, and registry keys" Sharif et al. (2019) before and after alteration. Although their testing was limited to a small subset of files, Sharif's publication of clear criteria for functionality verification was notable.

Lastly, the authors discuss several potential defenses, arguing that *code randomization* could be detected by reducing a binary to a minimal lexicographical representation, i.e., determining that the syntactical changes are semantically identical. *Code displacement*, however, could be made difficult to detect using opaque predicates Collberg et al. (1997), which make determination that functional bytes have been moved and non-functional code placed in their prior location difficult.

The paper by Lucas and Sharif Lucas et al. (2021) contains much of the same information, albeit with more detail in the appendix on tested defenses. Their final conclusion is that ML-based defenses are either too computationally expensive to deploy in a production environment or relatively easy to counter.

Song's paper focused on action minimization Song et al. (2020). They used stochastic hill-climb to find an episode (series of perturbations) that allowed malware to evade detection, then removed actions from this episode to find which actions were influential in shifting the label applied by the model. Actions that remained after pruning had their selection weights increased so Song's framework learned over time which perturbations were most effective. The actions they used were drawn from Anderson Anderson et al. (2017, 2018), with the exception of the *upx pack*, *upx unpack*, and *new entry point* actions due to concerns over functionality preservation. Note, the *overlay append* action they used draws content from benign files or random data with a given probability. The *code randomization* action was also used (discussed further in Section 4.3.5). Song tested a number of models: EMBER GBDT, ClamAV (open-source), and 4 anonymous AV tools. The most effective actions were reported for each target model. For the EMBER GBDT, *overlay append* was the most effective (contained in 80% of successful episodes). In terms of the EMBER feature space, this action affected the byte histogram, byte-entropy histogram, and strings. For ClamAV, the most effective actions were *benign overlay append*, *code randomization*,

random section append, *remove debug*. For the commercial AVs, the most effective action was *section create*, indicating a brittle reliance on the number of sections in a PE file Song et al. (2020). Song tested functionality using Cuckoo. They introduced a similarity metric $hits(G, G_{eva})/G_{count}$ where G and G_{eva} are the detected events from the original and altered samples, respectively. G_{count} is the number of events from the original sample while the $hits()$ function counts the number of times an event occurred in both the original and altered samples. If the results of this equation was greater than .8, Song labeled it as functional. They adopted this method due to observed variation in malware activity between executions. This is consistent with anti-sandboxing behavior that may be randomized. Since most authors do not discuss this, we theorize that other authors either ran their malware one time only, filtered out anti-sandbox files initially, or used a set of malware that did not contain anti-sandbox behavior to begin with. Song found that 91% to 98% of the binaries remained functional after alteration, noting that *code randomization*—as implemented by Pappas not Sharif—did not handle 16-bit instructions. In some instances, one of the Anderson actions used broke functionality. Song refers to "adding a section after the last section", but the caption for the error figure mentions "overlay append", so it is unclear whether *section create* or *overlay append* is the action responsible Song et al. (2020). If the action in question is *section create*, then the error would result from overwriting bytes that follow the last section table entry, which could occur if the last section table entry is right on the boundary with the first section contents or if code/data is hidden after the last section table entry but before the start of the first section. If the action is *overlay append*, it would depend on how the offset to the overlay (EOF) is found. If the section table is used to calculate the offset to the end of the file, any data hidden after the last section would be overwritten. Using the actual file size to find the EOF, however, would ensure that no existing bytes are overwritten. This might still result in an error if the PE file in question is a self-extracting Zip. A self-extracting Zip is a polyglot Koch et al. (2022) consisting of a Zip archive appended to the end of a PE file whose executable code unzips the

archive. In this case, adding content to the overlay might interfere with the unzip routine since the unzipping routine assumes content in the overlay is part of the Zip.

Song would later produce an RL-based version Song et al. (2021) of their action-minimization system (discussed in Section 4.3.4), but appears to compare the RL version to a random selection criteria rather than the stochastic hill-climb as used in this paper.

Zhong's paper contributed a truly novel set of instrumentation methods compared to prior work. These three method encrypt the executable section (*obfusmal*) or all (*stealmal*, *hollowmal*) of a malware file and alter the address of entry point to point toward a new decryption section. This is the state of the file when it is scanned by a static detector. Upon execution of the altered file, however, the decryption section decrypts the malicious content and either passes execution straight to the malicious content (*obfusmal*) or creates a new process (*stealmal*, *obfusmal*) for the malicious content before passing control to it. These actions alter the both the static features via encryption and dynamic features by altering the initial control flow. *Hollowmal* alters the static features the most by appending the malicious content and decypting content to an existing benign file and adding corresponding section table entries for the two new sections. Zhong's first paper, discussed here, used a GAN as the action selection mechanism. The generator did not generate the raw binary; rather, it selected which action would be applied to a given binary. Thus it's output was a vector of length three where each index mapped back to one of the three action. In the first paper, one version of each action was used, while in the second paper multiple variations of each action could be selected, depending on the value at that index. Both the generator and detector were CNNs whose feature vector consisted of the DLLs and system functions imported by each binary file. The target model was Virustotal, using an aggregate of detections from each vendor hosted on the site. Zhong found that their framework, named MalFox, reduced the average number of detections from 80.8 to 35.7. Zhong's second paper compared this GAN to a number of reinforcement learning models and is discussed further in Section 4.3.4.

Aryal's paper demonstrated an instrumentation method that, much like Zhong's *obfusmal*, leverages the address of entry point to undo a perturbation at the start of execution. Their *gradient section overwrite* method first copies an existing target section to the overlay. A small code loader (24 bytes) is inserted in the padding bytes of the .text section (conventionally, the executable section in a PE file). The final section's section table entry is adjusted (the raw size) so that the relocated data appears to be part of the last section. Next, the original location of the target section is overwritten with a gradient-optimized payload. The address of entry point is altered to point to the code loader, meaning the loader is able to perform actions before normal execution beings. In this state, the file is scanned by a static malware detector. When the file is executed, however, execution begins with the code loader, which moves the original section contents back to their original location. This action overwrites the payload, effectively reversing the perturbation and ensuring that the injected payload does not interfere with functionality. Control is then passed to the original address of entry point and the PE file executes as intended. Since payload bytes are overwritten by the original contents at the start of execution, they do not need to be functional. The payload must fit inside the raw size of the target section to avoid overwriting/relocating any other sections. There must be 24 bytes available in the padding bytes of the .text section. The perturbation does leave behind some evidence, namely the code loader (if the target section is not .text) and any changes to section permissions. The last section must be readable while the target section must be writable. The section containing the code loader must be executable. The authors compared their work to Kolosnjaji's *gradient overlay append* action and Kreuk's *FGSM section append* action against both MalConv and Malconv2, a larger capacity (up to 200MB as opposed to 2MB) version released in 2021 Raff et al. (2020a). Both models were only trained on 7000 binaries of which 6000 were malicious. They found that their attack achieved a maximum ER of 92% against MalConv by targeting the *.rdata* section (read-only data) and a maximum Er of 98% against MalConv2 by targeting the *.text* section (executable code). In contrast, Kolosnjaji's *gradient overlay append*

reached an ER of 16% against MalConv and 4% against MalConv2, indicating that the increase in capacity reduced the model's vulnerability to simple append attacks. Note, these authors budgeted three iterations for Kreuk's method instead of the 20 used in Kreuk's original paper. Next, the Aryal swapped the full gradient guidance of their methods for FGSM guidance in order to compare their work to Kreuk's prior method that used FGSM for guidance. In this test, Aryal's method reached a max ER of 96% against MalConv using *.rdata* as the target section compared to a max ER of 41% for Kreuk's method. Against MalConv2, Aryal's method reached an ER of 94% using *.rdata* as the target section while Kreuk's method reached an ER of 55%. Aryal's max evasions rates all occurred when using the maximum payload size (15% of file size) they allowed. The authors mention functionality preservation testing, but were unable to provide an assurance of functionality preservation. Their method depends on disabling address space layout randomization (ASLR) and involved filtering out PE files with obfuscated section names, as these names are indicative of custom packers that could deviate from format specifications.

This vein of research is quite diverse in terms of the actions implemented. Due to the diversity of methods, direct comparisons are rare, but summarized here.

Suciu directly compared *overlay append* actions, finding that FGSM outperformed full gradient optimization, random byte selection, and benign byte selection Suciu et al. (2019). Chen found that *FGSM overlay append* could be significantly improved by starting with bytes selected via GradCAM values extracted from a MalConv model. This gain is attributed to the sparsity of non-zero gradients when back-propagating through MalConv's max pooling layer Chen et al. (2019).

In black-box scenarios, an EMBER GBDT was not vulnerable to gradient-optimized bytes placed in the DOS header/stub area or appended to the overlay Demetrio et al. (2021c). This model was only vulnerable when chunks of bytes from benign files were selected and appended using genetic optimization. The use of a surrogate model (MalConv) to provide gradient updates during training did not yield attacks that transferred well to the EMBER GBDT Demetrio et al. (2021c). However, transfer

attacks have worked well against different architectures trained on the same feature space Rosenberg et al. (2018); Demetrio et al. (2021c). This indicates that the strength of the transfer property depends upon the similarity between a surrogate model's feature space and the target model's feature space. In lieu of similar feature sets, attacks created using a black-box genetic algorithm appear more effective than gradient attacks that leverage a dissimilar surrogate mode.

In terms of target models, MalConv is the most common target by far, followed by an EMBER GBDT and commercial models hosted on virustotal. MalConv's reliance on raw bytes paired with two known mechanism for reconstructing raw input from optimized features seems to make it particularly vulnerable, although all tested models displayed some level of vulnerability to adversarial samples. As Demetrio notes, the inclusion of non-functional bytes as features enlarges the attack space and lowers the skill level needed to generate an adversarial sample Demetrio et al. (2019b). While an EMBER GBDT was less vulnerable to overlay or DOS header/stub attacks, it is was hardly immune Demetrio et al. (2021c).

The target model of Zhang's paper—a graph neural network—is notable since it extracts a control flow graph as the feature set for each file. The control flow graph consists of nodes and edges that map out the possible execution path of a Windows PE file. Therefore, raw bytes are not included in this feature space. The only surveyed input-space method for affecting this feature set, aside from packing, would be the *code randomization* or *code displacement* actions used by Sharif and Lucas. Although Sharif's methods require more domain expertise than prior research (excepting Rosenberg's API wrapper), they demonstrate that even functional bytes can be altered. Moreover, these more intricate alterations are not readily detected via simple heuristics Sharif et al. (2019); Lucas et al. (2021), unlike *overlay append*.

Rosenberg, Sharif, Zhong, and Aryal demonstrated actions that affect static and behavioral features, the latter being quite rare. With regard to static features, Zhong's *stealmal* and *hollowmal* are particularly notable for encrypting the entire malware binary, drastically altering the static representation without using a common packer

like UPX. As indicated by Aghakhani Aghakhani et al. (2020), commercial detectors associate common packers with maliciousness, so custom routines like Zhong's have more real-world potential than simply relying on UPX.

For dynamic features, Rosenberg created an iterative reconstruction method that ensures only additive changes are made in order to attack a model trained on hybrid features. Sharif's instrumentation methods are the most sophisticated in the surveyed literature, weaving additional jumps into executable code and swapping out equivalent instructions. These two papers methods might point toward a future where behavioral features such as a control flow graph are as vulnerable as raw byte or structural features. However, a control flow graph-based model is still desirable since it would ignore perturbations based on non-functional bytes or structural elements of the PE file, which comprise the bulk of published methods.

With regards to functionality, Labaca-Castro and Song found that some Anderson actions (*new entry point*, *import append*, and either *section create* or *overlay append*) broke functionality in some malware files. Song also found that *code randomization* as implemented by *pappas* could damage functionality while Sharif—who created their own implementation—found it did not. Song proposed a similarity metric when evaluating results from a sandbox to account for anti-sandbox behavior that may result in malware presenting different behavior each time it is run. Rosenberg also discussed anti-sandbox behavior, opting to filter out malware that did present sufficient activity in a sandbox prior to training and testing their methods.

4.3.3 Algorithm Crossover Papers

Zhong was the only author to compare a gradient method with RL on the same action set Zhong et al. (2022). Likewise, Demetrio was the only author to compare a gradient guidance to genetic guidance Demetrio et al. (2021c). These comparisons only allow for limited conclusions regarding the efficacy of each approach.

Zhong et al. published a pair of papers. The first presented MalFox, a generative adversarial network (GAN) for guiding binary instrumentation Zhong et al. (2020). Unlike a GAN trained to generate images, this GAN did not generate a complete binary; rather, it picked from three discrete actions that altered the binary. Thus, the results cannot be applied to the prevalent *overlay append* or *section append* actions where gradient values are used to directly optimize bytes not select discrete actions. The second paper compared the performance of a variety of RL algorithms (acting on the same three actions) to this GAN Zhong et al. (2022). The results demonstrated that dynamic programming was the most effective algorithm, with temporal difference learning a close second. That DP was top performer is not surprising since it learns a complete model of the environment, i.e., it tests all possible action-state pairs. The other algorithms do not provide this guarantee. However, this exhaustive exploration carries a very high computational cost, limiting DP's applicability with larger state and action spaces Sutton et al. (2018). Temporal difference (TD) learning, however, uses Monte Carlo methods to sample the environment, allowing TD to remain computationally viable in more complex environments than DP Sutton et al. (2018). TD methods predate the development of deep RL models (DQN) for more complex environments, yet outperform DQN in this experiment with a small set of actions.

Table 4.1 reports the evasion rates of each guidance algorithm. The *single action* and *unaltered* rows feature no guidance whatsoever to establish baseline performance. VirusTotal was the target model for this experiment while the malware samples were drawn from VirusShare. Due to the uniqueness of the Zhong action set, it is difficult to compare the results to the other papers in this space.

In Demetrio's case, the genetic algorithm outperformed the gradient method. In this experiment, the GA selected byte chunks from benign files whereas the gradient method optimized (initially) random bytes. Since random bytes may not be significant to the model initially, sufficient non-zero gradient may not be available to effectively optimize Chen et al. (2019). A comparison between Chen's *FGSM benign append*

action and Demetrio's *genetic benign append* action would, perhaps, be a more direct comparison.

Song created multiple versions of their malware alteration pipeline, including pure random, stochastic hill-climb, and RL optimization methods. However, they do not appear to have compared the RL and hill-climb methods; instead they report comparing the RL method to the purely random version. Similarly, Labaca-Castro produced two version of their pipeline, one using a genetic algorithm and the other Distributed Double Deep Q-Learning (DiDDQN Horgan et al. (2018)), but did not report a direct comparison between the two versions.

4.3.4 RL Guidance Papers

A summary of this section is provided in Tables 4.6 and 4.7. Initially, RL guidance was only slightly better than randomly selected alterations, with Anderson scoring an *evasion rate* (ER) of 16% for RL and 13% for randomAnderson et al. (2017). Subsequent work improved the distinction between RL and random guidance. Chen Chen et al. (2020) scored 63% evasion with random guidance and 83% evasion with RL guidance. Wu Wu et al. (2018) found that Actor-Critic with Experience Replay (ACER Wang et al. (2016a)) (ER: 30%) did not outperform random (ER: 31%) guidance, while DQN Mnih et al. (2013) (ER: 48%), Double Deep Q-learning Network (DDQN van Hasselt et al. (2015)) (ER: 47%), and State-Action-Reward-State-Action (SARSA Sutton et al. (2018)) (ER: 47%) did. Quertier demonstrated that the REINFORCE algorithm Williams (1992) (80%) outperforms DQN (67%) and random guidance (25%) when tested on the same malware, action set, and target modelQuertier et al. (2022). This demonstrates that, given the correct algorithm, RL-guidance outperforms randomly applied alterations by a large margin.

Interestingly, Song Song et al. (2021) utilized a multi-armed bandit (MAB Sutton et al. (2018)) to minimize the number of alterations necessary for evasion, demonstrating that RL can be used to winnow an action set down to the most effective

alteration methods. Anderson Anderson et al. (2018), Wu Wu et al. (2018), and Y Fang Fang et al. (2020) further demonstrated that retraining a detection model on malware altered by RL improves the detection rate of the defending model. Anderson Anderson et al. (2018) reduced the evasion rate (ER) by 33%. Wu Wu et al. (2018) reduced the ER by 83%. Y Fang reduced their evasion rate by 84%, noting that the overall evasion rate (including malware not altered by RL) decreased by 57%. Thus, defensive retraining increased the overall performance of the defensive model.

Zhan Zhan et al. (2023) made a a number of novel contributions. First, they used Thompson sampling Thompson (1933) to select subactions rather than enumeration or randomization Zhan et al. (2023). For example, then using an *Overlay Append* action, the data appended was drawn from benign files in chunks, with the Thompson sampling procedure learning over time which chunks were most effective. This is the first paper to use a hierarchical optimization approach where the RL learns which action to select and Thompson sampling learns which content/subaction is optimal for that action. Second, Zhan conducted a feature importance analysis using Shapley values Shapley (1953) to determine which groups of features (data directories, general, sections, imports, exports, strings, byte histogram, byte-entropy histogram) were most significant to a GBDT trained on EMBER features, finding that the data directories were by far the most significant feature, at least according to Shapley. Shapley values have known limitations, especially the assumption that features are not correlated Kumar et al. (2020) which is untrue for EMBER features. The byte histogram, for instance, will change any time any other feature is altered. Third, Zhan augmented the the experience replay buffer of their RL agent using Shapley values. They first calculated the effect each Anderson action had on feature groups, recording the effect of each action on the top three features (i.e., the data directories features, general file features, and header features). When sampling from transitions, the RL agent (a Double Dueling Deep Q-Learning Network or D3QN Sewak (2019); Wang et al. (2016b)) prioritizes transitions that feature actions with a high Shapley effect. Against a target GBDT trained on EMBER features using binaries drawn

from the EMBER Anderson and Roth (2018a) and SOREL-20M Harang and Rudd (2020) datasets, this enhanced RL agent achieved an evasion rate of 78% and 72%, respectively. Individually, the Thompson sampling and Shapley prioritization increased the evasion rate by approximately 13% compared to a vanilla D3QN. Together, they increased the evasion rate by 28%, perhaps hinting at a synergistic effect Zhan et al. (2023). Zhan also compared their work to prior art, outperforming Anderson Anderson et al. (2018), Song Song et al. (2021),Y Fang Fang et al. (2021), Z Fang Fang et al. (2019) and Labaca-Castro Labaca-Castro et al. (2021).

Kozák Kozák et al. (2024) compared RL algorithms, finding that PPO outperformed DQN, policy gradient (PG), and random action selection against an EMBER GBDT. Interestingly, the random baseline agent was more effective against commercial detectors in a transfer attack than the trained RL agent, perhaps indicating that the commercial detectors did not rely on the same features as the EMBER GBDT against which the attacker was trained.

4.3.4.1 RL Actions

In order to keep the action set small and thereby simplify the learning task, Anderson utilizes random sub-selection in some of the actions Anderson et al. (2018). When a section is renamed, for instance, the new name is drawn uniformly from a list. This is an attempt to keep the action set small since RL models traditionally struggle when the state and/or action sets are large Sutton et al. (2018). Note, there are variations in naming and implementation of actions across papers. To facilitate comparisons between papers we have utilized a single naming convention across the papers reviewed despite likely yet subtle differences in implementation. A deeper examination of implementation differences is not always possible due to a lack of information in certain papers. Song and Anderson posted their code to a public repository, allowing for more detailed analysis of their action spaces.

Returning to the question of randomized subselection of actions, Chen Chen et al. (2020) and Y Fang Fang et al. (2020) argue that, since each sub-action may produce

different results, random sub-selection actually increases the learning challenge for the model. Chen, therefore, converted the Anderson action set to a completely deterministic one. In other words, a *Section Rename* action yields the same new name each time it is selected. Note, Chen omitted the *New Entry Point* action when replicating the Anderson set. This smaller deterministic action set outperforms Anderson's larger stochastic action set in a head-to-head comparison (83% evasion rate vs 65%) where all other variables are held constant.

Y Fang does not provide a head-to-head comparison, but does demonstrate that defensive retraining on RL-crafted malware improves the detection rate of a DNN defender. The overall evasion rate for all malware decreased from 1.93% to .83% while the evasion rate of RL-guided malware decreased from 19.13% to 3.1%. Y Fang utilized used five of the Anderson actions, but enumerated variants of some of them to create a total of 218 deterministic actions. Of these, 200 are *Import Append* actions that each add a unique function to the IAT. There are five variations of the *Rename Section* action and 10 of the *Section Create* action. The remaining three actions are *Break Certificate*, *Delete Debug*, and *Overlay Append*.

Wu Wu et al. (2018) is notable for adding new actions that manipulate more header fields to the existing eleven Anderson actions. Note, the paper is somewhat vague in describing the particular fields altered by some actions as well the values to which these fields are altered. What information is provided on the additional actions is relayed in Table 4.2. Wu outperforms Anderson in a direct comparison by 32.5% (47.5% evasion rate vs 15%). Wu also demonstrated that defensive retraining decreased the evasion rate of altered samples by 77.75%.

Z Fang Fang et al. (2019) demonstrated better performance than Anderson using only four actions that, unlike Chen's work, included stochastic sub-selection in their implementation. The four actions are *Remove Debug*, *Section Create*, *Overlay Append*, and *Import Append*. A head-to-head comparison of deterministic (Chen) and stochastic (Z Fang) action sets that both outperform Anderson is not available since the authors utilized different malware data sets sourced from Malshare and Virustotal, respectively.

Song Song et al. (2021) achieved better performance than Anderson in a head-to-head comparison utilizing seven of the Anderson actions and one from pappas, *code randomization*. The seven Anderson actions are *Overlay Append*, *Section Append*, *Section Create*, *Section Rename*, *Break Certificate*, *Remove Debug*, and *Break Checksum*. Song compared the functionality of malware altered by Anderson's action set implementation with malware altered by their own implementation and found that the Anderson actions damaged functionality of 63% of tested malware. Song's action set, carefully implemented to handle more edge cases, only damaged 8% of altered malware.

Quertier Quertier et al. (2022) slightly expanded on the Anderson action set by altering more non-essential fields in the header such as the time stamp. In addition, Quertier split *section append* and *overlay append* into two and threee actions, respectively, based on what kind of data was appended. Sufficient detail to authoritatively describe them is lacking. Zhan also made minor (in terms of number of features altered) additions to the Anderson action set that are detailed in Section 4.3.5.

Kozák focused on functionality preservation, finding that prior work did not preserve functionality Kozák et al. (2024). Out of a test set of 100 benign binaries altered via Anderson's *Random Section Create* action, only four still functioned. This reinforced Song's assertion that the Anderson action set is highly likely to damage functionality Song et al. (2020). The methods developed by Kozák show a clear improvement in functionality preservation with 98 out of 100 binaries altered by *Random Section Create* still functional. Kozák used a thorough methodology to verify functionality, executing each file three times to demonstrate reliable and detectable behavior prior to alteration. After alteration, each file was again executed three times. Clear criteria in terms of Cuckoo artifacts were published and a similarity metric similar to the one proposed by Song Song et al. (2020) was used to address seemingly minor deviations in behavior between execution runs.

With the notable exceptions of Wu, Y Fang, Song and Quertier, RL-based researchers have utilized smaller action sets than Anderson. Since smaller and larger action sets have both performed well, albeit on different data sets, we can not make a determination as to the ideal size for an action set. Although the deterministic actions used by Chen outperformed Anderson's stochastic action set, Song also outperformed Anderson using a stochastic action set. Unlike other domains with clear benchmarks for performance on common tasks Duan et al. (2016), for the malware instrumentation domain we can not yet determine the ideal algorithm with respect to the evasion rate. Finally, we note that RL papers tend to utilize instrumentation methods that are simple to implement. The instrumentation methods developed by Sharif and Zhong methods in particular (detailed in Section 4.3.5) show that more complex and comprehensive methods can be implemented. Thus we conclude that much work remains in the action space for this domain.

4.3.4.2 RL Algorithms

A wide variety of RL algorithms have been applied for the malware alteration task. A conclusive determination of the most effective algorithm and known-safe alteration methods has not been reached. Most papers do not conduct a direct comparison between their methods and prior work on the same malware data set against the same target detector. Successful evasion rates can not be directly compared between papers that utilize different action sets, malware data, and target models. Therefore, we only draw conclusions about the efficacy of a methodology when a direct comparison to prior work exists.

The foundational work by Anderson used a deep Q-learning network. Q-learning is an example of off-policy temporal difference learning, which means the model follows a stochastic (usually epsilon-greedy) policy while learning a deterministic policy. This allows for the exploration of all possible actions without negatively impacting the final exploitation phase Sutton et al. (2018). Function approximation via deep neural networks allows Deep Q-learning to avoid the combinatorial explosion characteristic

of tabular Q-learning when the action/state space is large. Neural networks also allow the model to generalize to similar states not reached in training Sutton et al. (2018); Mnih et al. (2013).

Actor-critic methods are a common choice in this area due to their efficacy in competitive scenarios Sutton et al. (2018). However, DQN has proven superior to Double dDep Q-Learning Networks (DDQN van Hasselt et al. (2015)), Actor-Critic with Experience Replay (ACER Wang et al. (2016a)) , State-Action-Reward-State-Action (SARSA Sutton et al. (2018)), and Advantage Actor-Critic (A2C Mnih et al. (2016)) when evaluated on the same action set and malware Wu et al. (2018); Chen et al. (2020). This is especially interesting in the case off DDQN, which was developed to counter an over-estimation error common in DQN models Sutton et al. (2018).

As discussed in Section 4.3.3, Zhong demonstrated that dynamic programming and temporal diffeerence learning outperform DQN and GAN on small actions sets.

Quertier conducted a head-to-head comparison of DQN, REINFORCE (a gradient-based method), and random action selection against 3 malware detectors: EMBER, MalConv, and an unnamed commercial off-the-shelf (COTS) malware detector. The action set and malware data set were held constant. REINFORCE significantly performed DQN and random action selection against EMBER and the COTS detector. With regard to MalConv, both RL algorithms achieved a 100% evasion rate. Interestingly, random action selection performed surprisingly well against the COTS tool. This sort of information on the real-world threat posed by this family of methods is extremely valuable. The results are contained in Table 4.3.

AMG-VAC (Adversarial Malware Generator – Variational Actor-Critic) also outperformed Double Deep Q-learning Network (DDQN van Hasselt et al. (2015)) and ACER in a direct comparison on this task Ebrahimi et al. (2021). By direct comparison, we mean one in which the action space, target model, malware data set, and all other parameters not specific to the RL algorithm are held constant. This comparison would allow future researchers to select the optimal RL algorithm for known action sets and target models. It is worth noting that on other tasks with

Table 4.1: Zhong Zhong et al. (2022) Detection Rates

Action Selection Method	Average **Detection** Rate
No Action: Unaltered Malware	68.8%
DQN	39.5%
Convolutional DQN	37.3%
Single Action: Obfusmal	36.2%
Single Action: Stealmal	30.0%
Single Action: HollowMal	29.8%
MalFox (GAN)	29.7%
Temporal Difference	27.5%
Dynamic Programming	23.2%

Table 4.2: Wu Wu et al. (2018) Novel Actions

Name	Description
Rename Time	Alters the date and time stamps in the header.
Version Reorder	Alters various fields, including the minor image version, in the optional header.
Resize Section	Modifies the *size of raw data* field of sections in the section table.
Reorder Resource	Shuffles the order of included modifiable resources. PE files can contain other files, such as images, as resources.
Modify Optional	Alter unimportant fields in the header, including the checksum.
Section Rebuild	Rebuilds a section.

Table 4.3: Quertier Results

	EMBER	MalConv	COTS Malware Detector
DQN	67%	100%	30%
REINFORCE	80%	100%	70%
Random	25%	76%	50%

a large state space VAE has demonstrated higher rewards than Soft Actor-Critic (SAC) and Deep Deterministic Policy Gradient (DDPG Fellows et al. (2019)). A larger state space allows for a higher-fidelity representation of the malware input, possibly increasing the real-world threat posed by RL-guided malware alteration. We recommend a direct comparison of DQN, REINFORCE, and AMG-VAC to determine which performs best with known action sets. It would be advisable to conduct this direct comparison multiple times with different action space cardinalities and target models to determine if one RL algorithm is optimal across all targets and action space sizes or if particular algorithms are most effective against certain combinations of action space and target model.

Wu Wu et al. (2018), Fang Fang et al. (2019), and Chen Chen et al. (2020) all held the action set constant when evaluating different RL algorithms, allowing for the conclusions drawn in Section 4.3.4.2.

Since RL algorithms are well-known to be sensitive to the size of the action space, holding an algorithm constant while varying the size of the action set would also be useful. For example, consolidating the work of Wu Wu et al. (2018), Z Fang Fang et al. (2019) by comparing their larger and smaller action sets with the original Anderson set would answer the question of which action set cardinality is most effective with a DQN agent. Chen Chen et al. (2020) and Y Fang Fang et al. (2020) already demonstrated that removing randomized action sub-selection improves agent performance.

4.3.5 Alteration Methods

This section focuses on the binary instrumentation actions implemented by all of the surveyed authors. A brief taxonomy discussion is provided to place binary instrumentation methods within the context of known obfuscation methods. This leads naturally to the discussion of functionality preservation, a necessary criteria for an adversarial binary that has not received sufficient attention. Following this, implementation details in terms of which bytes/structures are altered by an action

as well as the general feature set targeted (static or behavioral) are provided. The sections are broken down by guidance algorithm. Each action is described by a header of the form {Author} (static, behavioral, or both) {action name}. Names were adjusted in several cases due to overlapping or vague terminology (e.g., *padding*) in the original paper. Our first goal was to establish a common terminology for discussing these papers in reference to one another. Our second goal was to provide future researchers with a single source of previous work so that novel avenues can be uncovered. Where sufficient details are provided to contrast an existing action with a new variant of the action, we have included the new variant. A number of papers do not provide implementation details that would allow us to call attention to their novel contributions. We would encourage researchers to either provide detailed descriptions of their actions or post their code publicly so we can better highlight their contributions.

4.3.5.1 Taxonomy for Metrics and Methods

Collberg established a taxonomy for obfuscating transformations applied to digital files. Toward this end, he named three metrics of interest for each transformation: *potency*, *resilience*, and *cost* Collberg et al. (1997). First, the *potency* of an action measures how much complexity the action adds to the existing program. Second, the *resilience* of an action measures the cost of deobfuscating the file. Finally, the *cost* of an obfuscation measures the increase in computation resources needed to execute the obfuscated file Collberg et al. (1997). Given our context of altering malware to evade detection by ML models, we can be more specific about what these metrics mean when considering adversarial binaries.

In our context, *potency* would map to the *evasion rate*, which measures the percentage of files whose label (as applied by the target model) crosses the decision boundary from malicious to benign thanks to instrumentation. This is the most commonly utilized metric in surveyed papers. Typically, researchers have reported the overall evasion rate of an attacking system that selects from multiple instrumentation

methods rather than reporting the evasiveness of each instrumentation sans guidance. Quantifying the specific evasion rate of each action could provide useful information for researchers developing the next generation of instrumentation actions.

Returning to Collberg, the *resilience* of an alteration action is, in our context, difficult to directly measure. A target/surrogate model does not attempt to unravel the obfuscation; rather, it attempts to correctly label the sample despite the obfuscation. Thus, in our context the *resilience* of the action reduces to the *potency*, i.e., how much the action affects the maliciousness score applied by the target/surrogate to the altered file.

Since most of the action we will discuss are applied to structural rather then executable elements of a Windows PE file, a notable effect on the program's runtime is not discernible. Therefore, we instead consider the *cost* of an action to be the domain knowledge necessary to apply an obfuscating transformation without damaging the file's functionality. This necessitates another metric, *preservation*, which measures how likely the action is to preserve the functionality of the altered file.

4.3.5.2 Functionality Preservation

We assert that *preservation* is a significant metric for malicious adversarial samples. According to McGraw and Morrisett, malware is defined as "any code added, changed, or removed from a software system in order to intentionally cause harm or subvert the intended function of the system" McGraw and Morrisett (2000). A malware sample that has been fatally damaged can not cause harm or subvert any system. One can argue that broken malware is benign since it can not cause harm. Therefore, labeling a broken malware sample as an adversarial sample is theoretically incorrect. As mentioned earlier, Demetrio asserts that an adversarial sample is transformed "without disrupting its original behavior" Demetrio et al. (2021a). If the malware sample is no longer able to function at all, then the sample's true label has changed. This means that a large drop in the maliciousness score between the original sample and the broken adversarial sample is not necessarily evidence of a vulnerability in the

detector. Ergo, to claim that an adversarial sample is effective at tricking a model into applying the incorrect label, malware must remain demonstrably functional after transformation. In order to demonstrate functionality after transformation, one must first demonstrate that the malware completes a set of actions every time it is executed. Otherwise, loss of function will not always be detected or could be interpreted as the malware not executing of its own volition (anti-sandbox behavior).

The failure to preserve functionality might have implications for adversarial training. If the bulk of the adversarial samples upon which a model is retrained are broken, then an adversarially trained model might be learning to associate broken files with maliciousness rather than learning to recognize novel alterations. In our applied context of cybersecurity, perhaps this distinction is not relevant. It might even be advantageous to consider fatally damaged files as malicious if doing so improves the overall detection rate on known malware. If one considers a broken adversarial binary as part of a malicious campaign to alter the effectiveness of a target model, one could argue that the file is still correctly labeled as malicious despite the loss of functionality. Y Fang demonstrated that adversarial training not only improved the detection rate on adversarial samples but also on the whole corpus of malware and benignware Fang et al. (2020). It is possible the improvement came from labeling as malicious a number of broken malicious files in the corpus. Chen Chen et al. (2019) notes that adversarial training needs to be done carefully, finding that training on too many adversarial samples poisons the detector. Instead, they recommend filtering out adversarial samples using heuristics.

On the topic of functionality preservation, Y Fang notes that "Fortunately, as shown in experiments, the structure and function of each adversarial sample remained fine" Fang et al. (2020). Apparently none of the adversarial samples displayed a change in behavior. These authors evaluated the functionality of their adversarial samples using IDA Pro and Cuckoo. The first is a well-known reverse engineering tool and the second is a dynamic analysis tool or sandbox. The authors assert there was no change in the call graph or dynamic behavior of any adversarial sample when compared to

the original file. Note, the paper makes no mention of first checking that the malware yielded measurable behavior prior to alteration. An empty call graph does not change, so it is possible to make the above assertion concerning functionality preservation without actually demonstrating that the file completed the same nonempty set of actions before and after alteration. We make no assumption that this is the case, but call for more explicit phrasing as seen in Song's paper Song et al. (2021). Kreuk and Zhan make similar statements that there was no loss of functionality when using either IDA Pro or an unnamed tool to generate a control flow or behavior graph. None of these author, however, established that the graphs were non-empty or that the graph would change if a header field was invalid. Reverse engineering tools are designed to parse files despite malformation to enable the study of heavily obfuscated or intentionally neutered malware, so some demonstration that breaking a necessary header field would alter the reported control flow graph is warranted. Authors who used Cuckoo (e.g.,Kozák Kozák et al. (2024), Song Song et al. (2020)) to verify functionality found a large number of samples that were damaged using the same actions (though implementations might differ) as authors who found no damage but only verified using static graphs.

Song Song et al. (2021) points out that it is not unusual for malware to deviate from file format specifications by placing data at unexpected locations. Even with improved error handling via hidden data detection, Song does not report 100% *preservation* in altered malware. Thanks to edge case handling that included hidden data detection, Song was able to reduce the percentage of damaged files from 63% (unaltered Anderson actions) down to 8% (Song's improved actions). This result is highly significant since, according to Anderson, all of the actions he utilized preserved functionality on PE files.

Why then were 63% of malware damaged? First, Song notes that the library used for instrumentation, LIEF, can damage files simply by importing and exporting them Song et al. (2021). Second, malware does not always conform to file format specifications. In fact, malware authors are highly motivated to create files that exploit

implementation discrepancies to flexible parsing behavior. The PE file format contains a number of areas where data can be staged without compromising functionality Vuksan and Pericin (2011). For example, the DOS header contains optional fields; the entire DOS Stub is optional; code/data sections often contain padding bytes that can be overwritten; the end of the PE (overlay) can be written to without altering functionality. Lastly, there is often padding space between the end of the section table and the start of code due to file alignment. If there is data/code hidden after the last section table entry, then adding another section (*section append* action) would overwrite this data, breaking functionality. Song's code first checks if there is data present here before adding a new section. If data is found, the action is aborted. However, it should be noted that Song does not describe how many times the malware was executed in the sandbox. Randomized execution cancellation (anti-sandbox behavior) by the malware could account for some of the apparent failures to execute when evaluating Anderson's actions.

Kozák's work establishes a thorough standard for functionality verification to which future papers should adhere, with one exception. At the outset, they execute files in a dynamic sandbox (Cuckoo) to demonstrate that the files execute detectable behavior prior to instrumentation. The metrics collected are clearly defined: signatures, processes, and API calls. Files are executed multiple times to account for stochastic behavior that changes from run to run. After the file is instrumented, it is again executed multiple times to ensure that reported behavior is consistent after alteration. A similarity metric is used to account for discrepancies in behavior, with a clearly defined (95%) similarity threshold. Kozák Kozák et al. (2024) also provides a breakdown of prior methods, comparing the preservation rate of Anderson Anderson et al. (2018) and Song Song et al. (2020) actions alongside their own. Of the original 11 Anderson actions, only *Random Overlay Append* preserved functionality for all 100 benign binaries tested. Only four binaries altered via *Random Section Create* still functioned while only eight binaries altered via *Random Section Append* remained functional. Clearly the assumption that functionality would be preserved was invalid

even for benign files. Kozák was able to increase the functionality preservation rate to 98% for *Random Section Create* and 99% for *Random Section Append*.

Note, one limitation of Kozák's approach is the restriction of functionality testing to benign files. It remains to be seen what sort of obfuscations in malware files could interfere with Kozák's methods. Song's earlier functionality verification test was performed on 50 malware samples, finding that their improvements to the Anderson actions increased functionality preservation from 23% for Anderson's *Random Section Append* to 97% for Song's version. Anderson's *Random Section Create* action was improved from 23% to 93% functional. Although Kozák's slightly outperformed Song's in terms of functionality preservation on benign files, it remains to be seen if Kozák's methods are better at preserving functionality on heavily obfuscated malware.

4.3.5.3 Reinforcement Learning Actions

Anderson Anderson et al. (2017) demonstrated eleven actions in his inaugural paper. Many subsequent authors have utilized either a subset of these actions or created novel variants of Anderson actions. The Anderson actions rely on random selection when choosing what content to append/insert in contrast to gradient-based methods that typically optimize individual bytes. This section contains the Anderson actions in addition to any actions utilized in RL papers that were described in sufficient detail.

- Anderson (static) *Random Import Append*: This action adds an unused entry to the import tables. Since this action increases the size of the import tables, LIEF will create a new *.idata* section to contain the larger import tables and update the two affected pointers in the *data directory*. The altered tables are the *import lookup table* (ILT) and the *import address table* (IAT). These two tables are identical until execution begins, at which point the ILT is used to overwrite the IAT, replacing the name/ordinal reference to each function with that function's memory address. If the function belongs to a .dll not already imported, then

the *import directory table* must also be updated with an additional entry that points to IAT and ILT tables for the new .dll.

- Anderson (static) *Random Section Rename*: The name of a section is provided as an aid to debugging; it is ignored by the program loader. The name can be freely altered to any other 8 byte UTF-8 string. This transformation is an example of a seemingly meaningless alteration that could disproportionately affect an ML model due to feature extraction. The EMBER feature set, discussed in 4.3.1, bins section information according to the hash of the section name. Therefore, this EMBER feature can be changed via section renaming even though this field is meaningless to the program loader.

- Anderson (static) *Random Section Create*: This action adds another section table entry to the end of the section table. Since the header of a PE file is aligned according to the *file alignment*, there are often padding bytes between the end of the section table and the start of the first code/data section. If this space is greater than or equal to 40 bytes, a new section table entry can be added without disruption. Otherwise, room can be made for the new section table entry by pushing all of the content after the end of the section table to a higher raw offset. To prevent this from interfering with functionality, the *raw offset* field of each section table entry would need be updated. The action, as implemented by Anderson via the LIEF library, adds random bytes to the end of the file, calculating the appropriate *raw offset* and *virtual address* for this new content by parsing the existing section table entries. Note, since raw and virtual space are allocated via pages, the new *raw offset* and *virtual address* must be aligned according to the *file alignment* and *section alignment* fields. To calculate the new *virtual address*, the first step is to find the largest value gained by summing each section's *virtual address* with its *virtual size*. This value should then be aligned (padded) to be a multiple of the *virtual alignment* set in the *optional header*. Otherwise, the program loader will attempt to create a new

virtual space that overlaps with a previous page, which will break functionality if the two pages have different RWX permissions set. These permissions must be set using the *characteristics* field in the new section table entry. One problem this action does not address is that malware binaries are known to hide data in slack spaces, like the padding area between the end of the section table and the start of code/data. Song's version of this action cancels if hidden data is found here Song et al. (2021). This hidden data could be relocated to a higher offset, but reverse engineering would be needed to determine if/how the hidden data is referenced by the malware file during execution. Therefore, relocation of this hidden data is likely to damage functionality. When adding a new section table entry, the *number of sections* field in the *COFF header* will need to be incremented. If the new section table entry has a virtual size greater than zero, the *size of image* field will need to be increased by the *virtual size* of the new section, maintaining alignment with the *section alignment*. The *size of headers* field in the *optional header* will need to be aligned and updated if it was necessary to push the raw code/data to a higher offset in order to create the new section table entry. Note, this value is aligned using the *file alignment*.

- Anderson (static) *Random Overlay Append*: A simple action that appends random data to the end of the PE file without adding a corresponding section table entry. This data is ignored by the program loader. The term *overlay* refers to regions that are not mapped into virtual memory during loading.

- Anderson (static and behavioral) UPX Pack: This action packs the file using UPX, an open-source compression tool. Packing, aside from reducing the size of a file, disrupts the entropy and and byte histogram of a file, affecting a file's representation in the EMBER feature set without damaging functionality. It should be noted some packers, especially UPX, are now strongly associated with malware. Commercial malware detection tools might consider a UPX packed file

more suspicious than the original file Aghakhani et al. (2020); Demetrio et al. (2021a) even though packing disrupts the original file's bytes.

- Anderson (static and behavioral) *UPX Unpack*: If a file is already UPX packed, this action unpacks it. Otherwise it does nothing.

- Anderson (static and behavioral) New Entry Point: The entry point is the virtual address where the instruction pointer is placed once the PE has been loaded into memory. Since DLLs are not directly executable, the *address of entry point* is required only for .exe files. This entry point can be altered without affecting functionality by using a trampoline, a short section of code that jumps back to an original location. In other words, the *address of entry point* field is overwritten with a new address that points to a jump instruction in an executable section. The destination of the jump contains a second jump, which returns to the original *address of entry point* address. Execution then proceeds as normal, preserving the functionality of the altered file, in theory. This action slightly alters the initial behavior of the file, so it could affect dynamic features as well as static.

- Anderson (static) *Break Certificate*: PE files can contain a certification (digital signing) section designed to prevent alterations to the file. This certification is not required by default, meaning it can be removed from files without loss of functionality under normal operating conditions.

- Anderson (static) *Break Checksum*: The *checksum* field from the *optional header* exists to make post-compilation alterations detectable. The field can be intentionally overwritten without loss of function under normal conditions. Non-academic sources claim to have reversed the algorithm; if true, this field could be updated after alteration to make the alteration appear original to the file.

- Quertier (static) *Strings Overlay Append*: This action appends strings drawn from benign files to the overlay.

- Quertier (static) *Strings Section Append*: This action appends strings drawn from benign files to the padding space typically present at the end of section due to page alignment.

- Y Fang (static) *Enumerated Import Append*: This version of Anderson's *Import Append* action does not use random selection. Instead, 200 separate actions were created that each add one additional function to the set of imported functions.

- Y Fang (static) *Enumerated Section Rename*: This version of Anderson's *Section Rename* does not use random selection. Instead, five separate actions each target one of the standard section names (e.g., ".text") and insert an "m" character (e.g., ".mtext").

- Y Fang (static) *Enumerated Section Create*: This version of Anderson's *Section Create* does not use random selection. Instead, 10 separate actions each create a section with a specific name (e.g., ".mpatch0").

- Y Fang (static) *Random Section Append*: This action inserts data in the padding space typically found at the end of sections due to alignment. A random amount of random bytes are written into any padding space found. This version of Quertier's action is reported since it provides more implementation details (e.g., how the amount to append is chosen).

- Y Fang (static) *Remove Debug*: This action zeros out the virtual address and size fields that correspond to the debug index in the *data directories*.

- Zhan (static) *Thompson Modify Timestamp*: This action uses Thompson sampling to select from a set of modification to the *timestamp* field of the *COFF header*. This is an expansion of Quertier's earlier *Modify Timestamp* action that set the timestamp to a single value (993636360). It is unclear which

modifications are available for the Thompson sampling to choose from in Zhan's variant.

- Zhan (static) *Thompson Modify Machine Type*: This action uses Thompson sampling to select from a set of modification to the *machine type* field of the *COFF header*. The exact modifications are unclear as are the modification to the *machine type* made by the Quertier's earlier *Modify Machine Type* action which does not use Thompson sampling.

- Zhan (static) *Thompson Modify Optional*: This actions uses Thompson sampling to select from a set of modifications to the optional header. Though the description is a bit vague, these appear to include the *major linker*,*minor linker*, *major image*, *minor image*, *major operating system*, *minor operating system* fields of the *optional header*.

- Zhan (static) *Thompson Modify Machine Type*: This action renames a section using a name selected via Thompson sampling. It is unclear if the Thompson sampling selects the new name or which section to rename.

- Zhan (static) *Thompson Section Append*: Adds bytes to the padding space usually present due to page alignment at the end of a section. The bytes are chosen from benign file chunks using Thompson sampling.

- Zhan (static) *Thompson Section Create*: This action uses Thompson sampling to choose what content to add when creating a new section. It appears that the choice refers to what content to add (as it does in the *Thompson Overlay Append* action), but also possible that sampling is used to set section fields (e.g., section name, section virtual size, section permissions).

- Zhan (static) *Thompson Overlay Append*: This action uses Thompson sampling to select a chunk from a benign file that is appended to the overlay.

- Kozák (static) *Increment Timestamp*: This action increases the timestamp value by 500 days.

- Kozák (static) *Decrement Timestamp*: This action decreases the timestamp value by 500 days.

4.3.5.4 Gradient-based Actions

This section provides implementation details of the actions used in gradient-based papers and discusses which features they affect. Gradient-based methods are used, in most cases, to directly optimize bytes which are added to the file in some location. Zhong is a notable exception who used a GAN to select discrete actions rather than operate directly on the bytes Zhong et al. (2022). Rosenberg, likewise, used the gradient to select which APIs and strings to add to a file to evade detection Rosenberg et al. (2018) rather than optimizing byte values.

- Kolosnjaji (static) *Gradient Overlay Append*: This action, referred to as *padding* by Kolosnjaji and Demetrio, appends data to the end of a PE file. We renamed it to align with Anderson's terminology and differentiate appending data to the end of the file (overlay) as opposed to paddings areas at the end of each section. This action differs from the Anderson *Overlay Append* by appending gradient-optimized bytes rather than random bytes.

- Demetrio (static) *Gradient Partial DOS*: This action alters all of the bytes in the DOS header except the signature at offset zero ("MZ") and the all-important *e_lfanew* pointer at $0x3c$. Gradient-based optimization is used to select the new bytes that are written to the DOS header. The action is referred to as partial DOS since the DOS Stub which follows the DOS header is not altered.

- Demetrio (static) *Gradient Full DOS*: This action perturbs all of the *DOS header* and *DOS stub* except the magic number ('MZ') at offset 0x0 and the *e_lfanew* pointer at offset 0x3c.

- Demetrio *Gradient Extend DOS*: This action is an extension of the above *Full DOS* action that increases the space available for perturbation. To make this space available while preserving functionality, all file contents beyond the *DOS stub* are pushed to a higher offset. The *e_lfanew* pointer from the *DOS header* and each *raw offset* in the section table must be updated as well as the *size of headers* field from the *optional header*, which must be *file* aligned.

- Demetrio (static) *Gradient Shift Sections*: This action pushes all of the code/data sections to a higher offset, creating space that can be freely perturbed after the end of the header but before the new start of existing code/data. The *raw offset* field from each section table entry must be updated to preserve functionality.

- Rosenberg (static and behavioral) *FGSM API Hooking*: This action uses a novel API hook to allow for unconstrained gradient optimization. Instead of trying to fit additional function calls into an existing binary, this action patches a malicious binary's import address table so that calls to external functions are intercepted (hooked) by a wrapper using an open-source tool called IAT Patcher. This wrapper calls the external function as intended, but then adds additional function calls. These additional functions are drawn from a list of functions known not to mutate any states that might affect functionality. Rosenburg uses an iterative process where functions are randomly inserted and the gradient of a surrogate model checked. If the sign of the gradient–calculated using the surrogate model–indicates that the addition of the random function shifted the score of the malware sample away from being labeled malicious, then the function is added to the malware via the wrapper. This process continues until the sample has crossed the decision boundary and is labeled benign. The surrogate, like the target model, is trained on 140 API calls reported by the Cuckoo sandbox. By hooking external calls in this manner, functionality is preserved while allowing the gradient optimizer free reign in selecting which API calls to add. Adding a

wrapper to the file means that the static representation would change as well as the behavioral representation.

- Rosenberg (static) *FGSM String Append*: This action appends strings to a malware file. The strings are drawn from the 20,000 most common strings in Rosenberg's dataset. The process is the same as above (*FGSM API hooking*) but with strings instead of API calls. The location where the strings are appended can either be the EOF (overlay) or in a new section whose location is not specified in the paper.

- Kreuk (static) *FGSM Overlay Append*: This action appends data to the end of the file. The bytes are optimized using FGSM, which calculates the sign of the gradient rather than the full value.

- Kreuk (static) *FGSM Section Append*: This action appends data to the padding areas of existing sections. Executable sections are avoided. The data is optimized via FGSM.

- Chen (static) *GradCAM Benign Overlay Append*: This action uses GradCAM to determine which code/data chunks from a set of benign files most contributed to the benign label of said file. These chunks are then written to the end of a malicious file.

- Chen (static) *FGSM+GradCAM Benign Overlay Append*: This action uses FGSM to tune the benign chunks selected via the *GradCAM Benign Overlay Ammppend* action. This method was developed based on the observation that gradients calculated using MalConv must pass through a max-pooling layer which nullifies all but the most significant gradient values, meaning that gradient-based updates will struggle to improve unless the initial random bytes are highly significant already. By starting with chunks known to be significant according to GradCAM, gradient-based updates are more likely to improve the appended data.

- Zhong (static and behavioral) *Obfusmal*: This action encrypts the executable section of a PE file and adds a new executable section to the end of the file. The address of entry point is altered to point to this new section. When the file is executed, the new section decrypts the original executable section and then passes control to it. This process is complicated by the need to manually handle any relocations if the file is not loaded at the preferred address by parsing the relocation table and updating the pointers with the correct base address using functions from *kernel32.dll*.

- Zhong (static and behavioral) *Stealmal*: This action encrypts the entire malware file and adds it as a section to a host PE file. When the host is executed, it creates a suspended process, decrypts the malware section, and writes the malware into the new process. Control is then passed to the new process and execution of the original malware begins. Note, to write the malicious content to the new process, functions from *ntdll.dll* are called to locate the malicious content in virtual space and provide room for them in the new process. These functions are not intended for user-space, so the author loads the relevant .dll, finds the address of the function, defines an identical function in their code, then calls this new function using a pointer to the function in *ntddl.dll* instead. Relocations must also be handled by the decrypting stub in case the process created for the malware differs from the default base address.

- Zhong (static and behavioral) *Hollowmal* This action also encrypts the entire malware file, but appends it to an existing known-benign file. A second section for decrypting the malware section is also added. The address of entry point for the benign file is altered to point to the decryption section which, as in *stealmal*, decrypts the malware contents and launches them in a new process. As with *stealmal*, the virtual location where the host file has been loaded must be located and room created in a new process using *ntdll.dll*. Relocations must also be handled.

- Aryal (static and behavioral) *Gradient Section Overwrite*: This novel action relocates existing code/data from an existing section to a temporary location. This temporary location is the end of the file; however, the last section's raw size is increased so that the relocated content appears to belong to the last section. Gradient-optimized bytes are then written to the original section location along with a set of instructions known as the code loader that will execute before normal control flow begins. The *address of entry point* field from the optional header is changed so it points to the code loader. When the file is run, execution begins with this code loader, which grabs the original contents of the section from their temporary location and restores them to their original location, overwriting the adversarial bytes. Once the section has been restored, the code loader then jumps to the original address of entry point so normal execution can begin. This means that the section contents change between when the detector scans the file prior to execution and after execution begins. This action could affect both static and behavioral features since the code loader slightly alters the malware's behavior. This idea of preserving functionality by reversing a perturbation at the start of execution is a novel and thought-provoking vein of new research.

4.3.5.5 Genetic Algorithm and Other Guidance Mechanism Actions

A handful of papers have used genetic algorithms to guide the perturbation process Demetrio et al. (2021a,c); Castro et al. (2019a). This approach is quite flexible, allowing for the integration of functionality verification and diversity (via a similarity metric) as terms in a fitness function. Demetrio's actions that used a genetic algorithm for guidance are provided here; Labaca-Castro utilized a subset of the Anderson actions. Other guidance mechanisms include stochastic hill-climb, fully random selection, or drawing randomly from known benign files. The optimization mechanism for each action is provided in the title.

- Demetrio (static) *Genetic Benign Overlay Append*: This action appends data from from a set of benign files to the end of a malware file. The genetic algorithm selects which chunk of data to append. The GA seeks to maximize the change in label while minimizing the number of bytes added. Note, Demetrio refers to the padding bytes added to the end of existing sections in order to align with page boundaries as the slack space (according to their fourth figure Demetrio et al. (2021a)), reserving the term padding area for the end of the file. Microsoft Bridge et al. (2021a), however, refers to the alignment bytes after each section as padding bytes and labels the end of the file as the overlay. We favor Microsoft's definition due to its more precise language.

- Demetrio (static) *Genetic Benign Section Create*: This action first creates a new section table entry, then adds corresponding content drawn from a benign file. If the new section table entry would overlap with the first section, the raw offsets of all section are updated. This action increases the range of features affected by an append transformation since it not only changes the raw bytes (section content) but also header fields (e.g., section table entries). In other words, it has the potential to affect not only raw byte models (e.g., MalConv) but also a model trained on a header-focused feature set like EMBER, which extracts section names, virtual size, raw sizes, and permissions. The genetic algorithm chooses which chunks from benign files to append to the end of the file.

- Demetrio (static) *Benign Overlay Append*: This is referred to as *padding* action in Demetrio, but we call it *overlay append* to be more specific about where the content is added and bring it in line with other publications. The contents are not gradient optimized, but drawn directly from the section contents of known benign files.

- Demetrio (static) *Benign Section Create*: Adds section content from a known benign file and adds an entry to the section table that corresponds to the added content. Since the content is drawn directly from another file, there is no

optimization of the byte contents. Referred to as *section injection* in Demetrio paper; we used *section create* instead to pull it in line with Anderson terminology.

- Chen (static) *Random Benign Overlay Append*: This action selects random chunks from benign files to append to the overlay. Multiple chunks are appended until evasion or the model's input capacity is reached, with the results recorded as trajectories.

- Chen (static) *Experience Benign Overlay Append*: This action uses the trajectory information from the *random benign overlay append* action, selecting chunks according to how often they appear in successful trajectories.

- Suciu (static) *Benign Overlay Append*: This action appends content drawn from the headers of benign files to the end of a malware file. No optimization of the bytes is performed and the selection criteria is assumed to be random since it is not specified.

- Suciu (static) *Random Overlay Append*: This action appends random bytes to the end of the file, mainly for use as a baseline when comparing other methods of selecting bytes to append.

- Sharif (static and behavioral) *Code Displacement*: This action, adapted from a technique for defending against ROP gadgets Pappas et al. (2012), begins by conservatively disassembling a binary. Although complete disassembly is an open-challenge, IDA Pro and other tools can be run with all speculation based on heuristics disabled, yielding incomplete results (in terms of binary content coverage) that are highly reliable. These disassembled code snippets are then instrumented before being written back to the binary. The displacement action acts on basic blacks, short sets of instructions with only one entry and exit to avoid accidentally overwriting a valid jump command. Instructions of five bytes or more in length are located within basic blocks then and copied to a new location. The original location is overwritten with a jump command that

points to the new location. The commands at the new location are terminated by another jump command that returns execution to the next original command after the displaced code. Any gaps created by displacing the code are filled with semantic NOPs, which are instruction sequences that preserve all stateful elements like registers. Note, in a white-box scenario, the particular instructions used in the NOP sequence are chosen using the gradient. The new location could be in the padding bytes of an existing section or in a new section; in either case, the section needs appropriate permissions (read/execute). This is a rare example of an action that affects both the static and behavioral representation of a binary. Changing the bytes corresponding to instruction sequences alters the static representation, while adding jumps to new locations and sequences of semantic NOPs alters the fine-grain behavioral behavior of the file (basic blocks in control flow graph) to some degree. This action requires more domain knowledge and intermediary steps than the static-only actions that have dominated research in this space.

- Sharif (static and behavioral) *Code Randomization*: This action, also adapted from Pappas Pappas et al. (2012), has four subactions. This first is *equivalent code replacement*, which replaces instructions with semantic equivalents of the same length, e.g., replacing *add* 10 with *sub* −10 Sharif et al. (2019). The second, *register replacement*, finds all instructions that reference registers within a basic block and consistently swaps the registers. In other words, if *eax* and *ebx* are both altered by instructions, all *eax* operands are swapped with all *ebx* operands, provided no subsequent code references those registers without first reassigning their value. The third sub-action, *instruction reordering*, rearranges instructions within a basic block, using a dependency graph to ensure that no instruction is placed after an instruction upon which it depends. The last sub-action, stack reorder, rearranges the order in which register values are pushed to or popped from the stack for function calls. This set of sub-actions is one of, if not the,

most intricate alterations applied to a binary file. It alters not only the static representation but also the dynamic behavior of the binary. In the white-box version of this action, a valid sub-action is chosen for each function and applied to the binary only if the gradient shifts in the correct direction. In the grey-box version, the score output by the model is used to determine if a random function transformation is worth keeping (stochastic hill climb). As with the *code displacement* action, this action is a complex tactic that can affect both the static and behavioral representations of the binary, although it would depend on how fine-grain the behavioral representation is.

4.4 Conclusion

4.4.1 Limitations

Although a wide variety of algorithms and action sets have been tested, it is difficult to draw conclusions given the lack of consistency in terms of the malware data set, actions available to the optimization algorithm, target model, surrogate model (if applicable), scenario (white/grey/black), functionality verification tool/metric. Most papers focus on implementing a new instrumentation method or applying a novel (at least in this space) algorithm rather than comparing to prior work.

A comprehensive analysis comparing all previously used algorithms on the same data set and defensive model is warranted. Note, Zhan Zhan et al. (2023), Kozák Kozák et al. (2024), Wu Wu et al. (2018) and Ebrahimi Ebrahimi et al. (2021) took steps toward this comparative analysis by comparing four and three RL models, respectively. Demetrio conducted the only set of experiments to compare gradient and genetic approaches, finding that the genetic approach was more effective than a gradient method that relied on a surrogate with different features.

Some questions remain. For instance, is reinforcement learning, the most common optimization method, really the best choice? It is an unusual use case for RL since we

do not, necessarily, have a temporal delay problem. Genetic algorithms seem a more natural fit due to the immediate and discrete nature of the action space, but have not been thoroughly explored or compared to RL. Labaca-Castro produced both genetic and RL versions of their framework, but did not compare them. Zhong's paper was the only one to compare RL and gradient approaches, finding that dynamic programming and temporal difference learning outperformed a GAN when selecting from a small set of actions. This is an indication that RL is effective, but is not a conclusive result given the small action/observation space.

Little attention has been paid to adequately demonstrating functionality preservation (see Section 4.3.5.2). With the exception of Rosenberg and Kozák Kozák et al. (2024), most authors have not addressed how anti-sandbox behavior could interfere with functionality verification. While some authors—especially Rosenberg, Song, Kozák—have used clear metrics for functionality verification, most have not. That Song, Labaca-Castro, and Kozák Kozák et al. (2024) found widespread loss of function due to some action while other authors did not leaves lingering questions as to how thoroughly functionality has been verified even in papers that claim to do so. Note, it is possible that this difference in reporting is due to the level of obfuscation/malformation in the respective malware files used or implementation differences in individual actions.

Standard assumptions about how a file will execute are not always valid for malware that uses custom obfuscations. One area that appears to have been skipped in prior research is a determination of the level to which malware conforms to file format specifications. According to an outdated but oft-quoted presentation by Sun Systems, 68% of PE binaries are malformed Sheehan (2007); if this is true of PE files in general, what can we expect from malicious PE files? This is a question that should be addressed as a foundational step since file format specifications are used to design instrumentation actions and assert that these actions are functionality preserving.

4.4.2 Future Work

Moving forward, we recommend that authors use Rosenberg's twofold strategy of A) detecting and removing non-functional/anti-sandbox binaries prior to instrumentation so clear behavior can be detected and B) establishing a clear, specific metric (e.g., API calls) for functionality preservation on a specific tool (e.g., Cuckoo). Song makes a logical argument that this metric may need to use similarity rather than strict matching due to stochastic behavior in some files. This tool should be a dynamic analysis virtual machine that measures actual activity on a file system (e.g., Cuckoo) rather than a reverse engineering tool that is designed to attempt parsing despite malformation (e.g., IDA Pro, Ghidra).

4.4.2.1 Functional Malware Data Set

Malware availability is a persistent problem in cybersecurity; however, the adoption of a standard set of malware-to-be-altered would allow researchers to demonstrate concrete gains in performance (in terms of evasion rate and functionality preservation) over prior work on the same data set. Since malware evolves over time, however, this standard set would need to be updated periodically. The data set should possess the following traits:

- Preferred Format: The Windows PE is a common malware vector, an open challenge to reverse engineer, and the most widely studied format thus far. Since the Windows ecosystem continues to be widely used, this is the best candidate format in order to demonstrate gains over prior research.

- Known Function: The malware needs to exhibit known and detectable behavior to guarantee that transformation did or did not damage functionality.

- Determinism: The malware needs to execute the same way every time it is run in a common sandbox like CAPE so that any change in behavior can be blamed on an applied transformation rather than anti-sandbox behavior. A benchmark

value, based on experimentation, should be established specifying how many times a file needs to be executed before, at least on average, we can conclude that its executed activity does not change between runs.

- Diverse Obfuscation: The malware should contain documented examples of common obfuscations patterns—e.g., encryption, packing, or data hidden after the section headers—to determine if said obfuscation invalidates assumptions made during the transformation process, thereby causing loss of functionality.

- Diverse Objective: Samples should complete a wide variety of objectives to determine if certain malicious functionality is difficult to camouflage without compromising functionality.

A detailed study establishing benchmarks for functionality preservation would be of tremendous value. This study should A) verify functionality prior to alteration; B) repeat execution multiple times prior to alteration to detect anti-sandbox behavior; C) Use a dynamic tool like Cuckoo since this is based on actions observed on the file system; and D) run altered malware multiple times to verify loss of functionality is due to the alteration and not anti-sandbox behavior. This evaluation should be repeated on all of the prior actions to demonstrate their functionality preservation conclusively. A similarity metric as used by Song may be necessary to account for stochastic behavior by some malware.

Along these lines, a large-scale analysis of file format abuses in both benign and malicious files demonstrate which assumptions based on file-format specifications are actually safe. If researchers understood which abuses are common and why these abuses are tolerated during execution, more intricate instrumentation actions could be developed and the functionality preservation rate of existing actions could be improved. There are a number of questions this study could answer. For example, what percentage of malware has data hidden in the *DOS stub* or after the *section table* that is overwritten by existing actions? Do packing routines deviate from file format specifications? Can we develop improved actions that preserve functionality

while altering files that have hidden data in unexpected regions? Could this hidden data be duplicated or relocated?

If a malware file is broken during instrumentation, is the malware detector correct to label the file as benign since it can do no harm? If the file is genuinely broken, then the sample can not be labeled adversarial (see Section 4.2.1). That said, perhaps a broken file should be considered malicious if labelling it thusly produces a more effective detector. In this case the malicious activity is a meta-activity, namely data poisoning. The effect of defensive training using known broken malware should be compared to the effect of training using known functional malware to demonstrate whether breaking the file is in part responsible for the shift in the model's applied label.

4.4.2.2 Algorithm

A head-to-head evaluation of the top-performing RL algorithms should be conducted so that the ideal RL algorithm for this task can be demonstrated. Z Fang Fang et al. (2019), Wu Wu et al. (2018), Chen Chen et al. (2020), Ebrahimi Ebrahimi et al. (2021), Zhan Zhan et al. (2023), and Kozák all compared their results to prior work, allowing for the following conclusions to be drawn. Based on these papers, a simple DQN outperforms ACER, DDQN, SARSA, and A2C whilst AMG-VAC outperforms DDQN and ACER. PPO, REINFORCE and D3QN outperform DQN in a head-to-head comparison. DP outperforms TD and DQN on a very small action space, but may not be computationally feasible on a larger space Zhong et al. (2022). Therefore, the next set of researchers should focus on either using REINFORCE, PPO, AMG-VAC, and DP (where feasible) or prove a new method is more effective than these four algorithms. In order to prove a new method is more effective, all other aspects of the test should be held constant: data set, action set, feature set, target model.

An evaluation of the computation cost to apply DP to a larger state and/or action space would also be useful. If the results of this DP experiment are promising, some

exploration of the smallest effective observation space should be conducted to widen the applicability of DP to the detection evasion problem.

Finally, it is currently unclear whether larger or smaller action sets are more effective once an RL algorithm has been chosen. Likewise, stochastic and deterministic variations on the Anderson action set have both performed well. An evaluation that holds the RL algorithm and malware data set constant while evaluating these variations would be useful.

Gradient methods, assuming a surrogate model for gradient extraction, have also shown promise but are typically used to optimize content rather than select an action. Perhaps a hybrid approach would be useful. Consider the simple *overlay append* action from Anderson's action space. Instead of appending random bytes or bytes drawn from a benign file, why not append bytes optimized by a gradient approach? The RL agent or a genetic algorithm could still be used to select the best action, but the action itself would be optimized by the gradient-based method. This would require the use of a surrogate with a similar feature space to the target model.

Genetic programming seems promising yet under-explored. An experiment or set of experiments that swaps RL for GA but holds all other values constant should be conducted.

Gradient-based approaches are largely focused on optimizing content, so they do not necessarily compete with RL/GA methods that select discrete actions. That said, a GA could be constructed that optimizes bytes rather than selecting actions. In this case, it would be productive to compare the GA to a gradient-based method. The GA would not require a surrogate for training, so it brings some advantages despite the computational cost.

4.4.2.3 Target Models

A comprehensive analysis of target models needs to be conducted to determine the level of vulnerability for each type of classification (defender) model. MalConv has been thoroughly attacked and its weakness well documented. EMBER appears more

resilient, but relies on structural elements can be altered without loss of function. A graph-based model would ignore many of the methods published this far, with the exception of actions that pack the binary or edit the executable code (Sharif, Lucas). This approach has not been tested by input-space instrumentation papers as of yet, although a feature-space alteration paper did explore evading GNNs Zhang et al. (2022). Continuing effort should be made to evaluate COTS tools, or at the very least Virustotal to determine which actions are still effective against COTS AV detectors in the current year.. Although several papers submitted their altered malware to Virustotal, other research indicates that submissions to Virustotal should be repeated periodically to authoritatively demonstrate effectiveness clearly Zhu et al. (2020). Testing of COTS tools should also continue to determine if these models are less susceptible than the COTS versions hosted on Virustotal.

4.4.2.4 Feature Importance Recommendations

An attacking model—whether it uses RL, GA, or the gradient—is only as effective at shifting classification if the discrete transformations or vector of freely alterable raw offsets it is provided aligns with the significant features of the target model. Given the large potential search space (all functionality preserving alterations to a variable size binary) for novel actions, development should follow a principled approach. Rather than looking at which binary instrumentation actions are readily implemented, researchers should determine which features of the binary are significant to a target model.

Once researchers know which features are most significant to a model or family of models, they can then invest further time and resources in developing and testing functionality preserving perturbations that target those features. Demetrio conducted such a study prior to their action development, analyzing MalConv (raw byte features) to find byte regions that were both significant and freely alterable (or could be made to be freely alterable with a prior transformation like *extend* and *shift*). This led to their gradient-based method for targeting and extending the *DOS stub* and *DOS*

header. For example, Oyama conducted a feature importance study of EMBER using Shapley values. Their finding was that the import table was the most significant feature. Zhan also conducted a Shapley analysis of an EMBER GBDT, finding that A) the data directories were the most important feature Zhan et al. (2023) and B) Shapley was useful for action prioritization during training Zhan et al. (2023).

Models trained on raw bytes provide the simplest mapping from significant features back to alterable input. The EMBER features, on the other hand, bin some features using a hashing function (section table entries) and represent others as summary statistics (byte and byte-entropy histogram), so a mapping from significant features back to relevant perturbations of the input needs to be created.

On the other hand, the *DOS stub* is not directly represented in the EMBER feature space, so Demetrio's DOS header/stub focused attacks are not very effective against an EMBER model Demetrio et al. (2021c). The bytes contained within this area are only—along with the rest of the file–represented in the byte histogram and byte-entropy histogram features that EMBER extracts, diluting the effect of perturbing this region to some degree. A target model trained on the control flow graph, as demonstrated by Zhang Zhang et al. (2022), would not include the contents of the *DOS stub* at all, so perturbing this area would have no effect.

4.4.2.5 Action Space

Finally, much room for improvement remains in the action space domain. Sharif and Lucas conducted the most intricate actions, weaving their perturbations into the executable code section. Aryal demonstrated a reversible method for altering a section's contents. These methods indicate that perturbation is by no means limited to simple, easy to detect methods like *overlay append*. Although the bulk of prior research is focused on structural alterations, Sharif and Lucas demonstrate that executable code which generates the behavior of the file can be altered without loss of function.

It is possible to edit more locations within a binary using binary rewriting tools Schulte et al. (2022b), although we are not aware of any adversarial binary papers

leveraging these more advanced tools. Transforming the entire binary, as a packer or Zhong's framework Zhong et al. (2022) does, is another area where static detection tools may have further vulnerabilities. Since packing alters the control flow, these methods may be effective against behavioral models as well, although this has not been well explored yet. Demetrio demonstrated that a common assumption in other papers (that existing sections can not be relocated) is incorrect, leading one to wonder what other transformations have been assumed impossible. Another comprehensive change not well explored in surveyed literature would be rewriting the binary to execute in an obfuscated virtual machine that is customized for that individual binary, as done by commercial tools like CodeVirtualizer Technologies (2024). Note, this commercial product requires source code rather than compiled binaries, which is not available for the bulk of existing malware. For more reading on malware and virtual machines, see Sharif et al. Sharif et al. (2009).

As mentioned in Section 4.2.6.1, a reconstruction method for extracted features—e.g., the EMBER feature set—would allow for gradient-based attacks on models that do not rely on raw bytes. This mapping would require an intimate knowledge of PE file format specifications, tolerated format deviations, and malware obfuscation mechanisms (to avoid loss of functionality). This is not a trivial undertaking, but would allow for a definitive comparison between gradient-based alterations methods, RL methods, and genetic methods on extracted feature models that hithertofore have only been attacked via black-box optimization or surrogate models trained on raw bytes. This reconstruction method would need to be additive (e.g., Rosenberg's method for adding API calls) to prevent loss of functionality. Some fields would also need maximum bounds to conform to realistic values (e.g., maximum of 96 sections in PE files Bridge et al. (2021a)).

Table 4.4: Gradient and Genetic Papers

First Author	Paper Title	Year	Algorithm	Action Set	Target	Contribution	Limitations
Rosenberg	Generic Black-Box End-to-End Attack Against State of the Art API Call Based Malware Classifiers	2018	Gradient (FGSM)	FGSM API hooking FGSM string append	DNN, CNN, RF, SVM GBDT, LR RNN, GRU LSTM	Demonstrated actions effective against static and hybrid models (static and behavioral) Verified functionality using API call sequences detected by Cuckoo before and after alteration Demonstrated transferability against novel architectures and feature sets	Behavioral target model had only 1 feature: the API sequence that was altered Hybrid target model had only 2 features: API sequence and strings
Kolosnjaji	Adversarial Malware Binaries: Evading Deep Learning for Malware Detection in Executables	2018	Gradient	gradient overlay append (original name: padding)	MalConv	First to demonstrate gradient-based approach to altering bytes in binary First to demonstrate *closest positive* input reconstruction mechanism for mapping features through non-differentiable embedding back to binary files	Single action that only affects overlay Did not verify functionality
Kreuk	Deceiving End-to-End Deep Learning Malware Detectors using Adversarial Examples	2019	Gradient (FGSM)	FGSM section append (original name: mid-file injection) FGSM overlay append (original name: end-of-file injection)	MalConv	Demonstrated gradient-based method that wrote to padding areas of existing sections rather than overlay (EOF)	Did not report tool used to generate behavior graph
Demetrio	Explaining Vulnerabilities of Deep Learning to Adversarial Malware Binaries	2019	Gradient	partial DOS	MalConv	Demonstrated use of explainability metric (integrated gradients) to find regions that are significant to label, yet easy to manipulate without loss of function First to target beginning of file rather than append to end or between sections	Did not verify functionality Single action only targeted at raw byte models like MalConv
Chen	Adversarial Examples for CNN-Based Malware Detectors	2019	GradCam FGSM Random	GradCAM benign overlay append (original name: benign features append) FGSM+GradCAM benign overlay append (original name: enhanced-bfa) random benign overlay append (original name: random method) experience benign overlay append (original name: experience-based method)	MalConv	Demonstrated use of GradCAM to select benign data/code to append *GradCAM benign overlay append* Compared to randomly selecting data/code from benign files (*random benign overaly append*) Instead of initializing their gradient method (*gradient benign overlay append*) with random bytes, they start with code/data chunks selected by *GradCAM benign overlay append* action Similarly, *experience benign overlay append* selects the most successful data/code found by the *random benign overlay append* action Demonstrates effectiveness of adversarial training Demonstrated adversarial example detector Argued that blindly retraining on adversarial samples can poison data set	Approach specific to CNN architecture of MalConv model Only appends data to overlay Did not verify functionality
Sucio	Exploring Adversarial Examples in Malware Detection	2019	Gradient (FGSM)	benign overlay append random overlay append gradient overlay append from Kolosnjaji FGSM overlay append from Kreuk FGSM section append from Kreuk	MalConv	Compared effectiveness of append actions on MalConv Tested how scale/stratification of training data affected MalConv's performance Compared full gradient with FGSM, which only uses the sign of the gradient	Limited novelty in action set Did not verify functionality
Labaca-Castro	AIMED: Evolving Malware with Genetic Programming to Evade Detection	2019	Genetic	9 actions from Anderson (excluded imports append, create entry)	GBDT ESET Sophos Kaspersky	Verified functionality using Cuckoo Incorporated functionality into GA fitness function Demonstrated that GA learned to preserve functionality Compared to randomized system Demonstrated transferability of attacks from one model to another Tested commercial detectors (ESET, Sophos, Kaspersky)	Author later developed AIMED-RL, but did not compare RL to this prior GA work
Sharif	Optimization-Guided Binary Diversification to Mislead Neural Networks for Malware Detection	2019	Gradient (white-box) Stochastic hill-climb (grey-box)	code displacement (relocates code chunks using trampoline) code randomization (replaces instructions with semantic equivalents) Kreuk actions	AvastNet MalConv Endgame	Novel actions conservatively disassemble binary, relocate or replace instructions, then reassemble binary Compared actions to Kreuk actions Tested against MalConv and 2 commercial models (AvastNet and Endgame) Verified functionality of subset of binaries manually and using Cuckoo Demonstrated sanitization defense against Kreuk	Functionality preservation only tested on 10 benign and 10 malicious files

Table 4.5: Gradient and Genetic Papers Continued

First Author	Paper Title	Year	Algorithm	Action Set	Target	Contribution	Limitations
Song	Automatic Generation of Adversarial Examples for Interpreting Malware Classifiers	2020	Stochastic hill-climb	code randomization 8 actions from Anderson (excluded upx pack, upx unpack, new entry point)	GBDT ClamAV 4 COTS AVs	Verified Functionality Discovered instances where code randomization and Anderson actions break functionality Developed metric for testing functionality that accounts for stochastic behavior	Limited details on effectiveness of individual actions, e.g., which ones were not effective
Demetrio	Functionality-preserving Black-box Optimization of Adversarial Windows Malware	2021	Genetic	benign overlay append (original name: padding) benign section append (original name: section injection)	MalConv GBDT Virustotal	Demonstrated that altering section table more effective than merely appending bytes Demonstrated that black-box methods only slighly less effective than white-box Demonstrated that *section injection* action transferred well to commercial anti-virus (on Virustotal)	Functionality preservation not validated
Lucas	Malware Makeover: Breaking ML-based Static Analysis by Modifying Executable Bytes	2021	Gradient (white-box) Stochastic hill-climb (grey-box)	code displacement code randomization Kreuk actions	MalConv AvastNet Endgame Virustotal	Tested against multiple research models along with Virustotal Demonstrated Sharif action with gradient guidance in white-box attack Verified functionality	Limited novelty compared to Sharif
Demetrio	Adversarial EXEmples: A Survey and Experimental Evaluation of Practical Attacks on Machine Learning for Windows Malware Detection	2021	Gradient (white-box) Genetic (black-box)	gradient extend DOS gradient shift sections gradient full DOS genetic benign overlay append Kolosnjaji actions Kreuk actions	DNN MalConv GBDT	Demonstrated novel actions targeting DOS header and DOS stub Demonstrated that actions effective against models trained on raw bytes (e.g., MalConv) do not transfer well to GBDT trained on EMBER features	Only 1 action effective against GBDT: (*padding*) action from Demetrio set Functionality preservation not validated
Zhong	MalFox: Camouflaged Adversarial Malware Example Generation Based on Conv-GANs Against Black-Box Detectors	2022	GAN	obfusmal, stealmal, hollowmal	Virustotal	Demonstrated generative adversarial network (GAN) for altering malware binaries Tested against commercial malware detectors hosted on Virustotal Demonstrated novel actions that comprehensively alter binaries	GAN simply chooses variants/combinations from the 3 discrete actions defined Did not verify functionality
Aryal	Intra-Section Code Cave Injection for Adversarial Evasion Attacks on Windows PE Malware File	2024	Gradient	gradient section overwrite (original name: code cave injection)	MalConv MalConv2	Demonstrated first novel action that dynamically restores altered content First to test MalConv2, which has higher input capacity than prior MalConv model	Functionality preservation validation limited to checking for debug events MalConv (target model) vulnerabilities already well-documented

Table 4.6: Reinforcement Learning Papers

First Author	Paper Title	Year	RL Algorithm	Action Set	Target	Unique Contribution	Limitations
Anderson Anderson et al. (2017)	Evading Machine Learning Malware Detection	2017	DQN	Anderson (described in Section 4.3.5)	Gradient-Boosted Decision Tree (GBDT)	First to demonstrate that a reinforcement learning agent using functionality-preserving actions (perturbations) can alter malware files so that they evade detection by a machine learning model	Assumed functionality preserved Did not optimize content
Anderson Anderson et al. (2018)	Learning to Evade Static PE Machine Learning Models via Reinforcement Learning	2018	ACER	Anderson	GBDT	First to demonstrate that retraining a target model (GBDT) on adversarial samples (malware altered by an RL agent) improved the target model's ability to detect new adversarial samples created using the same methodology	Assumed functionality preserved Did not optimize content
Wu Wu et al. (2018)	Enhancing Machine Learning Based Malware Detection Model by Reinforcement Learning	2018	ACER (actor-critic with experience replay) DQN DDQN SARSA	Anderson++ (manipulates more header fields, vague details on 16 actions)	LightGBM GBDT	Demonstrated that increasing the number of actions available to the RL agent improved the evasion rate Used the same data set as Anderson Also compared the performance of RL algorithms, finding that DQN outperformed ACER, DDQN, and SARSA in terms of evasion rate Reinforced that defensive retraining reduced the evasion rate	Assumed functionality preserved Did not optimize content
Z Fang Fang et al. (2019)	Evading Anti-Malware Engines With Deep Reinforcement Learning	2019	DQN	Anderson - - (only 4 actions)	GBDT	Demonstrated that decreasing the number of actions available to the RL agent increased the evasion rate, perhaps contradicting WuWu et al. (2018) Trained on the same data set as Anderson First to verify functionality of adversarial samples (altered malware) via Cuckoo and IDA Pro	Did not demonstrate functionality prior to alteration Did not demonstrate IDA Pro would detect loss of function Did not optimize content
Chen Chen et al. (2020)	Generating Adversarial Examples for Static PE Malware Detector Based on Deep Reinforcement Learning	2020	DQN A2C	Anderson -	GBDT	Found that removing randomized subselection of actions from the Anderson action set improved the evasion rate Found that DQN outperformed A2C in terms of evasion rate	Assumed functionality preserved Did not optimize content
Y Fang Fang et al. (2020)	DeepDetectNet vs RLAttackNet: An Adversarial Method to Improve Deep Learning Based Static Malware Detection Model	2020	D3QN	Anderson++ (218 actions)	8-layer fully-connected deep neural network	Removed random sub-selection from Anderson action set. Verified functionality of adversarial samples via Cuckoo and IDA Pro Reinforced that defensive retraining on adversarial samples reduced the evasion rate	Did not demonstrate functionality prior to alteration Did not demonstrate IDA Pro would detect loss of function Did not optimize content
Ebrahimi Ebrahimi et al. (2021)	Binary Black-Box Attacks Against Static Malware Detectors with Reinforcement Learning in Discrete Action Spaces	2021	AMG-VAC DDQN ACER	Anderson -	EMBER (Gradient Boost) Malconv (Convolutional Neural Network)	Utilized variational inference (VI) AMG-VAC outperformed DDQN and ACER Verified functionality using virustotal	Did not demonstrate functionality prior to alterationn
Z Fang Fang et al. (2021)	A3CMal: Generating adversarial samples to force targeted misclassification by reinforcement learning	2021	A3C	Anderson - -	Two different XGBoost (distributed gradient boost) models	Attempted misclassification of malware by family	Data set contained no benign files Only verified functionality with small subset Vague functionality criteria Did not optimize content
Labaca-Castro Labaca-Castro et al. (2021)	AIMED-RL: Exploring Adversarial Malware Examples with Reinforcement Learning	2021	DiDDQN	Anderson -	LightGBM (light gradient boosting)	Verified functionality using Cuckoo Encouraged exploration noisy nets Utilized a more complex reward function to encourage diversity	Did not optimize content
Li Li and Li (2021)	An IRL-based malware adversarial generation method to evade anti-malware engines	2021	IRL-DQN	Anderson - -	Virustotal	Training parameters discovered via inverse RL DQN model trained using the discovered parameters	Did not verify functionality Did not optimize content
Song Song et al. (2021)	MAB-Malware: A Reinforcement Learning Framework for Attacking Static Malware Classifiers	2021	MAB	Anderson+	Malconv Ember	Verified functionality using Cuckoo Demonstrated pruning method to find minimal effective alteration per file	Did not optimize content

Table 4.7: Reinforcement Learning Papers Continued

First Author	Paper Title	Year	RL Algorithm	Action Set	Target	Unique Contribution	Limitations
Gibert Gibert et al. (2022)	Enhancing the Insertion of NOP Instructions to Obfuscate Malware via Deep Reinforcement Learning	2022	DDQN	NOP Insertion	CNN	Attempted misclassification of malware by family	Did not attempt to shift label from malicious to benign Did not verify functionality
Zhang Zhang et al. (2022)	Semantics-preserving Reinforcement Learning Attack Against Graph Neural Networks for Malware Detection	2022	DQN	28 variations of NOP insertion	DGCNN (Deep Graph Convolutional Neural Network) GCN (Graph Convolutional Network)	Targetted behavioral model trained on control flow graph	Altered features not actual malware
Zhong Zhong et al. (2022)	Reinforcement learning based adversarial malware example generation against black-box detectors	2022	DQN Convolutional DQN Temporal Difference Dynamic Programming MalFox (GAN)	Obfusmal Stealmal Hollowmal (described in Section 4.3.5)	Virustotal	Encrypted malicious code, adding sections/files to decrypt and transfer execution via three distinct actions	Did not verify functionality
Quertier Quertier et al. (2022)	MERLIN – Malware Evasion with Reinforcement LearnINg	2022	DQN REINFORCE Random	Anderson+	MalConv EMBER COTS malware detector	Compared performance of DQN, REINFORCE and random guidance on same action set Evaluated unknown commercial off-the-shelf (COTS) malware detector	Did not verify functionality Did not optimize content
Zhan Zhan et al. (2023)	PSP-Mal: Evading Malware Detection via Prioritized Experience-based Reinforcement Learning with Shapley Prior	2023	DQN MAB DQEAF A3CMAL Genetic AIMED-RL	Anderson+-	GBDT	Omitted Anderson actions *new entry point*, *UPX pack/unpack*, *break certificate* Added actions *modify timestamp*, *modify optional fields*, *modify machine type* Optimized content using Thompson sampling Prioritized action replay via Shapley priors	Did not demonstrate functionality prior to alteration Did not demonstrate IDA Pro would detect loss of function
Kozák Kozák et al. (2024)	Creating valid adversarial examples of malware	2024	DQN PG PPO Random	Anderson+-	GBDT Virustotal	Omitted Anderson actions *new entry point*, *UPX pack/unpack* Added actions *increase timestamp*, *decrease timestamp* Compared performance of DQN, PG, PPO guidance on same action set Demonstrated loss of function in prior work Demonstrated improved function preservation Thoroughly evaluated functionality	Did not optimize content

Chapter 5

Advancing Static Binary Instrumentation to Improve Evasion Rate

5.1 Introduction

Our hypothesis, based on the survey, was that the actions available to an attacking RL agent are a major limiting factor in prior research. To test this, we proposed to develop a novel action and integrate it into an RL agent while holding all other test parameters constant. The performance of the agent would be measured by comparing the evasion rate (ER) of malware files altered by the baseline agent with the ER of malware files altered by our improved agent. The survey also revealed that verification of functionality preservation is lacking in most papers and a necessary test in order to assert that the true label of an altered file is still accurate. Therefore, we asserted that our novel action must be thoroughly tested for functionality preservation. This chapter details our experiment to validate our hypothesis and is divided in to the following sections:

1. Explainability-Driven Development

(a) Shapley Methods

(b) Shapley Results

(c) Novel Action Development

2. Functionality Preservation

(a) Functionality Criteria

(b) Relevant PE Structures

(c) Evaluating Functionality Preservation of Prior Actions

(d) Novel Action Variants to Address Obfuscation

After testing whether our action increased the evasion rate, we then dove into an ablation study to understand which features most contributed to the increased evasion rate and to confirm whether our action development process was optimal. By development process we refer to our use of information gain and Shapley values as feature importance metrics. Since the search space of possible binary instrumentation methods is quite large and validation of functionality preservation is time-consuming, we did not blindly experiment with possible instrumentation actions. Instead, we used information gain and Shapley values to estimate feature significance and designed our novel action to target these features.

The ablation study demonstrated that this method, while it did yield an evasive action, was not optimal. However, the ablation study revealed a significant feature which we then targeted using another novel action. This second action scored an extremely high evasion rate (100% on our small functionality preservation set and 97% on a larger random sample). Note, this second action did not require the use of an RL agent or any other actions to reach this high evasion rate. Instead, it used a surrogate model paired with a reconstruction method. This chapter details the process of how we arrived at this highly evasive action.

5.2 Explainability-driven Development

The survey revealed that the MalConv model has been thoroughly evaded by prior efforts. Since the model is differentiable and translation invariant, *Overlay Append* (see Section 4.3.5 for nomenclature details) attacks are highly effective against it. Three methods have also been published for mapping from the feature space to the raw bytes, simplifying the task of crafting adversarial binaries that can evade MalConv. We therefore focused on an EMBER GBDT as this seemed a more difficult and, therefore, interesting target. Since a GBDT is not differentiable, researchers either need a surrogate model trained on similar features or a black-box explainability method to find effective (in terms of evasion rate) perturbations. When developing our first novel action we followed the explainability approach rather than the surrogate model approach for the following two reasons.

First, Demetrio found that using MalConv as a surrogate for EMBER did not yield high evasion rates Demetrio et al. (2021c). The transfer property would suggest that the attacks transfer since both models are trained on the same objective Papernot et al. (2016a). However, the survey we conducted points out that the transfer property is dependent to some degree upon a similarity in the feature space. As a trivial example, we point out that a model trained on a control flow graph will completely ignore data appended to the overlay since these bytes are not mapped into a process and play no role in execution. Similarly, MalConv has an upper limit to its raw capacity, which means that bytes appended to a file whose size equals the model capacity will not be extracted Raff et al. (2018a). Therefore, these bytes can not affect the label applied by MalConv. The degree to which feature spaces must overlap in order for attacks to transfer, however, is unknown. Although Demetrio's transfer attack was not highly successful, we note that EMBER does have some sense of the raw bytes thanks to its byte histogram feature which counts the number of times each possible hexadecimal value occurs in the file. We will return to the topic of the byte histogram later.

Second, surrogate model methods require a reconstruction method to map from the feature space back to the input space. This reconstruction method is not trivial for the 2381 EMBER features which involve lossy binning of extracted features via the hashing trick. Simplifying assumptions would need to be made to map the binned features back to functional input. While a mask and bounds could be used for individual features, rules would also need to be generated that prevent a combination of features that are valid individually from combining to create a non-functional file. From a developmental perspective, it would be far more tractable if we could first determine which features are significant and then only write the reconstruction methods necessary to safely alter these features. To determine which features to focus on, we need some indicator of significant, i.e., a feature importance metric which Shapley values can provide.

5.2.1 Shapley Methods

We utilized results from a concurrent project at ORNL that focused on explainability for malware detection models. This project used information gain in conjunction with Shapley values to find significant features for an EMBER gradient boosted decision tree, our target model. This method provided us with an indication of where to start development of our novel action and the ablation study we conducted afterward provided insights into which features an EMBER GBDT considers significant.

During the learning process a decision tree chooses features for nodes based on how well they split the data between classes. This split can be measured using the GINI index or information gain. Information gain calculates the expected entropy of using a X feature at Y node Quinlan (1986). This allowed us to select the most discriminative features rather than analyzing the full $2,381$ EMBER features.

Subselection was used to avoid the computational cost of Shapley analysis on the full feature set. Shapley analysis provides the marginal contribution of each feature to the final label produced by the target model Shapley (1953). This contribution

is calculated by comparing the model's performance with and without each feature, which requires evaluating the target model 2^N times where N is the number of features Fryer et al. (2021).

The use of Shapley values for explaining a GBDT trained on EMBER features has precedent. Zhan used Shapley values to prioritize actions chosen by an RL agent during training Zhan et al. (2023), finding that this prioritization improved the final model's evasion rate against EMBER compared to the baseline agent. Note, Zhan clustered the EMBER features by logical group (byte histogram, section features, general features, etc) before running the Shapley analysis whereas the ORNL analysis used information gain to subselect discriminative features prior to the Shapley analysis. This distinction in methodology may account for differing results.

As noted by Kumar, Shapley analysis has limitations as an explainability metric since it treats features as independent players when in reality features may be correlated Kumar et al. (2020). For the EMBER feature set, correlation between features is apparent. For instance, the byte histogram and byte-entropy histogram will change any time a byte is changed in the file, which means that adding a section table entry does not solely affect the section features but also the byte histogram and byte-entropy histogram feature. Even within feature groups like section features, the addition of a section affects numerous indices because all attributes of the section must be set at once. Therefore, the section name, section virtual size, section raw size, and section permission features will all change at the appropriate index each time a section is added, assuming that the sizes or permissions are not set to zero in the new section. Based on the section content that is added, other features may also change, e.g., the number of strings, entropy of strings, file length. Therefore, we utilized the Shapley results in designing our first action but conducted an ablation study afterward to determine if the Shapley values from the reduced feature set were a reliable indicator of significance to the full model.

5.2.2 Shapley Results

Examining the Shapley results in Figure 5.1, we saw some interesting patterns. The exception table was reported as the most significant feature. This maked sense as structured exception handling is a known method for malware to gain malicious control. However, interfering with this abnormal manipulation of the exception table would, in our estimation, require more reverse engineering than desired for automation of our method. We therefore examined the next set of features, which are extracted from section table entries. In the EMBER feature set, section data (raw size, virtual size, name, permissions) are each binned into vectors of length 50 using the hashing trick. The hashing trick exploits intentional hashing collisions to map variable-length input to a fixed-length feature vector. This trick is used for section data since the section count of a PE file may vary from 0 (for a highly abnormal malware file) to the specification maximum of 96.

We can see that section names which map to index 22 are the next most significant feature, but are not monotonic. We want our method to push the file toward a benign label monotonically so that repeated selection of the novel action does not negate the benefit of a prior selection. Moving on to the next features, we see that the virtual size is a significant feature at eight indices. Moreover, it is monotonic for indices 19 and 37. Positive values at index 19 and negative values at index 37 are both associated with maliciousness. If we could flip the sign of these individual features, we might be able to trigger misclassification. To determine how to do this, we examined EMBER's feature extraction process. The section table entries in a PE file are extracted via Scikit-learn's FeatureHasher function Pedregosa et al. (2011) which relies on a signed version of the Murmurhash3 algorithm. For the raw and virtual size features, the value stored is the raw or virtual size with the sign determined by the section name. For the name hashes, the value stored is either 1 or -1. The sign flips to prevent accumulation of the values in each index.

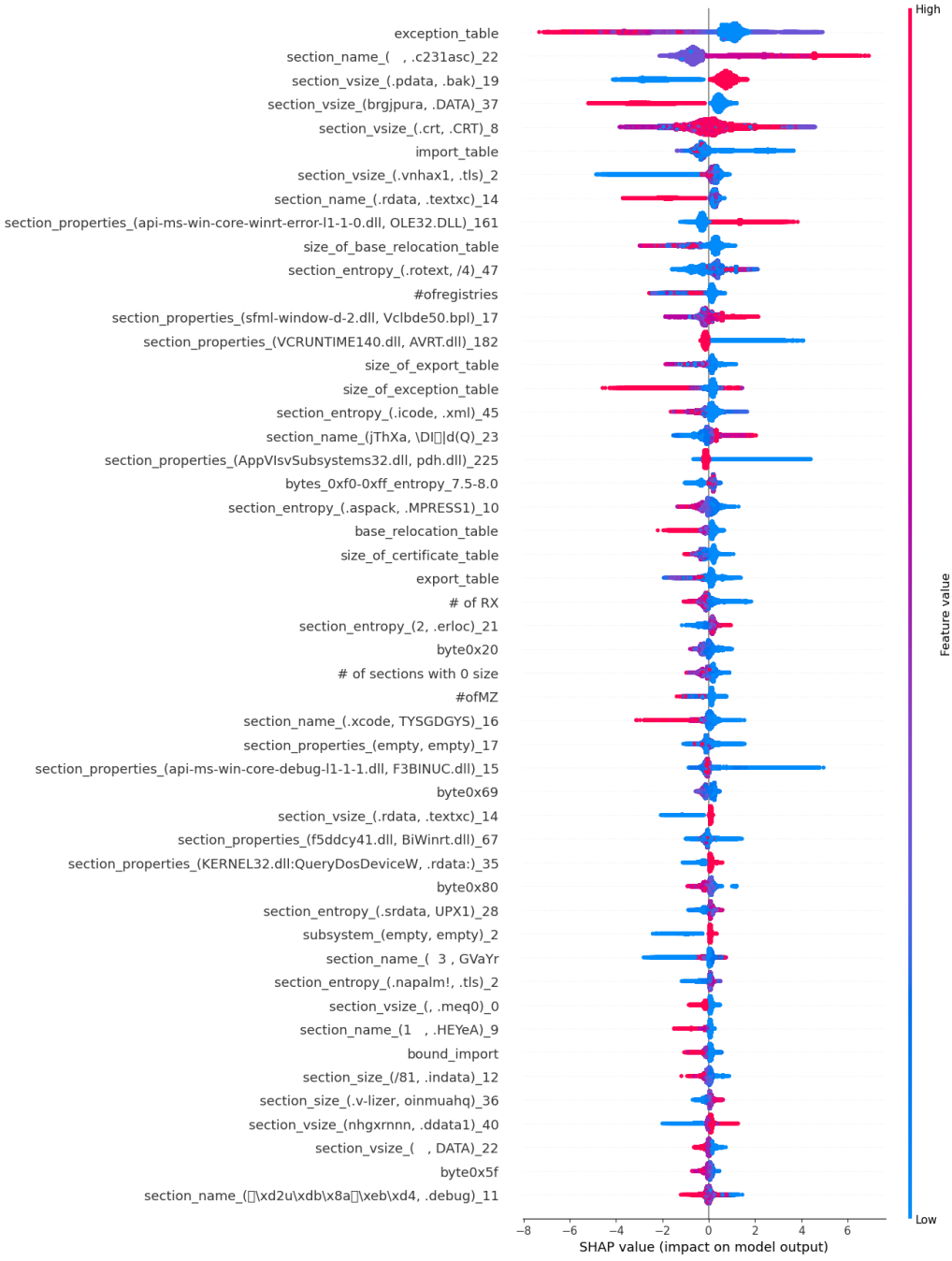


Figure 5.1: Shapley Results: We first used information gain to reduce the feature set. Next, we trained a model on this reduced feature set. Finally, we conducted a Shapley analysis to study the contribution of each feature index in the reduced feature set. We saw that section names and virtual sizes are the most significant, at least according to this analysis, features that are not a part of control flow (unlike the exception table).

However, this allows us to alter the feature without altering existing sections. Suppose a malicious PE has a positive value for the virtual size at index 19. Now suppose we add a section table entry with a larger virtual size whose name maps to a negative value. The value stored in index 19 will be now be negative. Therefore, we can flip the sign of the feature without changing the virtual size of an existing section. To do this, we need to know the sign that will be assigned based on the section name. Although hashing algorithms are intentionally irreversible, we were able to circumvent this by collecting section names from our database of malware, passing them individually to the EMBER feature extraction code (https://github.com/elastic/ember) that relies on Scikit-learn's FeatureHasher, and recording the resulting sign. We built a dataset of sorted function names that provides for each index a set of known positive and known negative names. When we add a section, we can draw from this pool of names so that the added section flips the sign of the desired index.

5.2.3 Novel Action: Section Virtual Negation

Based on the Shapley analysis, we created four variants of an action which we call *Section Virtual Negation* that targets the virtual size feature for sections whose name maps to indices 22, 37, 14 & 22. These actions alters the features at the given index without altering existing sections by adding a new section that also maps to the targeted indices, but has the desired sign. For sections 22 and 37, this is a positive sign while the desired sign is negative for sections 14 and 22. A new section is created by adding a section table entry to the end of the section table, increasing the *section count* and *size of image* fields, and finally appending the section contents to the end of the file. This action is distinct from prior section creation actions since it targets the section name and virtual size using the Shapley results as opposed to randomly assigning the name and picking the virtual size arbitrarily. Furthermore, we found malware obfuscations that interfered with functionality and found solutions to these obfuscations that have not been published before. An example of the effect this action

has on the feature space is presented in Figure 5.3, which shows how the hashed virtual size of section's whose name maps to index 14 is altered by our binary instrumentation action.

Although our goal was to affect the section names and section virtual sizes, section table entries and their corresponding content need to be added as a consistent (table entry values match content) unit in order to preserve functionality. This means that other features such as the byte entropy histogram and section table permission hashes are affected by the addition of a section. To increase the evasion rate of the action, we experimented with different values for each section attribute to improve the effect on these other features.

The raw offset, raw size, virtual size, virtual address, permissions (read/write/execute), and raw content must be set when adding a section. We utilized a pair of tried and true method for finding optimal values: fuzzing and Thompson sampling. We varied the raw and virtual size together via a nested loop. The step size for the virtual size was $0x1000$, the most common value for virtual pages in our dataset. The step size for the raw size was $0x200$ which is the default value for raw pages as specified by Microsoft Bridge et al. (2021b). We varied both values from $0x0$ to one megabyte, finding that a virtual size of $0x8000$ and a raw size of $0xa00$ were most effective. We repeated the test with Thompson sampling, and found the same values after sampling for $50,000$ iterations, indicating that in future Thompson sampling could be used instead of exhaustive fuzzing.

Next, we fuzzed the raw content. Since fuzzing every byte possible byte value for a section with even a small raw size would be computationally expensive, we appended a single byte N times where N was the raw size of the section. We found that $0x32$ was the most effective byte value.

The permissions (memory protections) for the added section were the final consideration. If we add a single section to file and alter only the permissions, we can see the effect of different permission combinations by comparing their effect on the maliciousness score. In order from most to least malicious, the permissions

are as follows: read/write/execute, read/execute, write/execute, read/write, write, execute, read. According to these results, the target model has learned to associate read/write/execute (RWX) memory with maliciousness, an insight that aligns with human analysis. RWX sections allow malicious authors to rewrite and then execute section contents after the file has been loaded into a process (virtual memory), allowing for functionality that may differ from the static representation already scanned by a malware detector. Based on these results, we set the permissions to read-only. Since the bytes we add to the file are not meant to alter execution, we have no need to set further permissions. Song's method for adding sections (discussed further in Section 5.3.3) only allows 4% of files to cross the decision boundary even when using the same virtual size, raw size, section name, and contents as our action because they set the permissions to read, write and execute rather than read-only.

5.3 Functionality Preservation

5.3.1 Functionality Criteria

We began testing with a random set of 317 malware files from Virushare. We executed the files six times to determine which files reliably demonstrated detectable behavior. Rosenberg was the first to demonstrate functionality prior to alteration Rosenberg et al. (2018) but subsequent authors have inexplicably not replicated this step. We argue this is a significant oversight because the maximum number of files which actually demonstrated detectable behavior in our sandbox during a single run prior to alteration was only 191 files. If we had proceeded without checking for functionality first, then for 126 files we could honestly claim no detectable loss of functionality even if we broke the file ($0/0 = 0$), which would be misleading. A number of prior papers claim functionality preservation without clearly stating that they verified functionality prior to alteration.

Another flaw in prior research regards the functionality verification oracle. We used the CAPE sandbox, a fork of Cuckoo, to verify functionality. This tool executes each file on an isolated VM and captures artifacts during said execution. Since this tool can report static and dynamic artifacts, we restricted ourselves to the following dynamic artifacts: files read/written, mutexes created, registry keys read/altered. Some prior authors are not clear in terms of which metrics they collected from a sandbox. Other authors used a reverse engineering (RE) tool as an oracle for functionality instead of a sandbox. We argue this is a flawed methodology since RE tools are designed to parse malware despite malformation. It is not uncommon for malware authors to damage a file in-transit to evade detection and then reconstruct the file later in the attack chain. For this reason, RE tools will attempt to determine the likely control flow of the file without validating that the file could execute. As a trivial demonstration of this, we can take a normal benign file and change the permission on the only executable section to 0. This means the file is mapped into memory without the ability to execute at all. Ghidra, the open-source RE tool released by the NSA, reports no change in the control flow graph of the benign file after this alteration. Therefore, RE tools are not—nor were they ever designed to be—reliable oracles for functionality. Furthermore, there are tool which claim to be a sandbox, but are in fact an emulator that predicts dynamic behavior based on static attributes (see OPSWAT's documentation ADD reference). This tool would also not be a reliable oracle since those static features are alterable via static binary instrumentation.

To be sure of functionality, we executed each file a total of six times before alteration, finding that only 61 exhibited the same level of activity on all six runs. Kozák argued in favor of repeated executions Kozák et al. (2024), which were clearly warranted by the high degree of variability in reported behavior of our malware samples. This repetition reduces the chance that we will mistakenly labeled as broken an altered file that is merely not activating of its own accord. Malware authors have been known to execute a simple Bernoulli experiment to determine if the malware file will execute on

a given run, a rather crude way to reduce the likelihood of detection in a sandbox. For this reason, a single execution may not be a reliable indicator of functionality.

We proceeded with this small set of files (n= 61) because A) it seemed to exhibit reliable behavior and B) executing each file in a dynamic sandbox is a time-intensive task that must be completed after each minor alteration to a binary instrumentation action. This focus on a small set also allowed us to hone in on specific file format specification violations that informed our development process.

As some prior authors have done, we ran UPX unpack on each file prior to instrumentation. This strips one layer of obfuscation away, simplifying our task. If a file is not UPX packed no change is made to the file. We do this since UPX deviates from file format specification by placing a UPX header in the padding space. This also allowed us to better examine the unique traits of each file, which are obuscated when a file is UPX packed. ADD discussion of UPX function. This UPX unpack and filtration of files with highly variable behavior are the only steps taken to simplify the task at hand, and are based on prior work. It is not uncommon for malware to be packed multiple times by different or custom packers, so this step does not guarantee that a file is no longer packed. The BEE67 file, for example, was packed by DWING, a free but not open-source packer which also violates file format specification.

We note that, even after collecting only the files that exhibited the same activity on six runs, we still had some variability during functionality verification testing. This highlights an issue with prior work that asserts no loss of function. If executing unaltered files multiple times results in the majority of malware files (261/317) behaving differently, how can prior assertions of full functionality preservation post alteration be accurate? Loss of function should have been detected as a result of unaltered behavior even if no alterations were applied to the file.

We emulate Song and Kozák by using a similarity score when evaluating altered files. While Song only executed files once using an 80% similarity threshold, Kozák checked three features after three execution runs. If two or more features were 95% similar in any of the three runs, then Kozák considered the file functional. We simplify

their method a bit, executing altered files three times and considering a file functional if 95% of the features match on at least one execution.

5.3.2 Relevant PE Structures

Although the survey in chapter 4 covers the PE format specification in more detail, relevant background on the PE structures our action interacts with are provided here. The complete structure of a PE file's headers is provided in Figure 5.2.

PE files begin with the *DOS header* which contains the *DOS signature* at offset $0x0$ and the *e_lfanew pointer* at $0x3c$. These two fields are the only components of the *DOS header* required for functionality and must be present at these exact offsets. The rest of the PE headers are found at the raw offset pointed to by *e_lfanew*, making this pointer critical. It must point to the rest of the header for functionality to be preserved. This also means that the entire NT header could be relocated if the *e_lfanew* pointer was updated correctly.

The *NT header* begins with the *PE signature* and contains the *COFF header* and *Optional header*. From the *COFF* header the only relevant fields are the *time/date stamp* which is $0x8$ bytes from the *PE signature* (will be relevant later) and the *number of sections*. When adding a section, the *number of sections* field needs to be increased, up to a maximum of 96.

The *Optional* header—ironically required for executable files—contains critical fields for mapping the file into a process, and is followed by the *data directories* and *section table*. A PE file is transformed from its on-disc (raw) representation to a its virtual representation using fields from the *optional header* and *section table*. The contents of a PE file are stored in sections, and referenced via the section table. The section table specifies the the raw offset and size of each section as well as the virtual offset and size of the section once it is mapped into a process.

Memory is allocated in pages, albeit in different amounts, for both the raw and virtual representation of the section. The *optional header's file alignment* field specifies

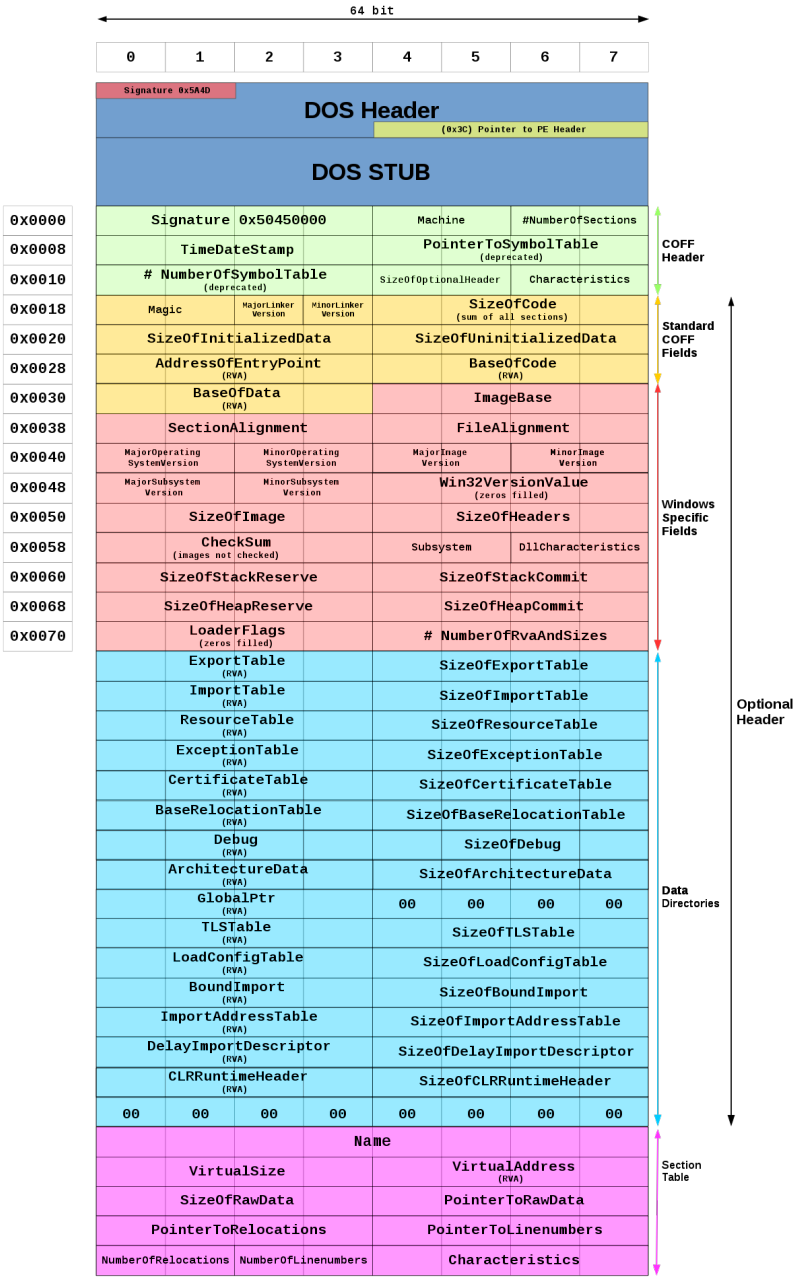


Figure 5.2: PE headers: The DOS header is the first structure in a PE file, followed by the DOS stub. Next we have the NT header, which contains the PE signature, COFF header, and optional header. This is followed by the data directories and section table. Finally, we have padding bytes from the last section table entry to the first section's contents.

the on-disc (raw) page size. The default size is $0x200$ Bridge et al. (2021b). The virtual size of a page is specified by the *section alignment* field and is typically $0x1000$. The difference in page size results in virtual and raw size rarely being the same value. In packed files, the difference can be quite stark. The UPX packer, for instance, typically creates two sections. One section, call it the unpacker, contains the compressed data and the unpacking stub. The other, call it the destination, has a raw size of zero (empty on-disc) but a large virtual size (in-process). As the unpacking stub executes it places uncompressed data from the unpacker section into the destination section's process space, gradually filling the process space without writing back to the raw file. This example is provided to illustrate how the two spaces interact and point out that the raw and virtual sizes of each section are not inherently similar and that large differences between the two may be present in benign files.

The required memory protections (permissions) for the pages the section will occupy in a process are set in the section table. These protections are designed to limit malware's ability to rewrite either itself or another process during execution and must be set when the pages are allocated. The permissions are any combination of read, write, and execute (R/W/X).

The *size of image* specifies the amount of virtual space to allocate for the PE file's entire process. It is a section-aligned value. The *size of headers* field specifies how far to scan within the file in order to extract the entire section table and is a file-aligned value Bridge et al. (2021b). The *optional header* includes the *data directories* which we will not touch upon except to note that these directories are referenced via virtual address not raw offset. This means that changes to raw offsets will not invalidate the *data directory entries*. The *section table* follows the *data directories*. Each entry has a name, raw offset, raw size, virtual offset, virtual size, and permissions field. The remaining fields are not required.

Since the headers are file-aligned, padding (zeros) bytes may be present after the last section table entry to ensure that the first section begins on a new page. We refer to adding these padding bytes in instrumentation actions as either *file-aligning*

(for raw offsets) or *section-aligning* (for virtual addresses) content. Note, *file-aligning* results in padding bytes actually being written into the file, while *section-aligning* takes place only in the process space, so no change is apparent in the file contents. Padding bytes in the raw file can be safely overwritten without loss of function. If 40 or more padding bytes are present, a new section table entry can be added to the section table by overwriting these 40 bytes. If sufficient bytes are not present, all section contents can be pushed to a higher offset as long as their section table entry's raw offset field is updated. The new raw offset must be a file-aligned value to ensure that the relocated sections still begin on new pages.

Note, we have not discussed 32 vs 64 bit thus far. We note that the only relevant difference aside from the larger addresses is that the optional header is larger in 64 bit. However, the `Pefile` library we used to parse the headers automatically handles the increase in size and addressing so it did not pose a challenge.

5.3.3 Evaluating Functionality Preservation of Prior Actions

From the Shapley analysis, we decided to target the virtual sizes of sections. To affect this feature without interfering with an existing section, we need to add a new section whose name will hash to the same index as the targeted existing section. Therefore, our *Section Virtual Negation* action builds on previous actions that added a section to a PE file (*Random Section Create*) or performed limited header relocation *Gradient Extend DOS*. The actions enacted by prior authors are discussed and tested for functionality preservation here with an emphasis on implementation details that interact with malware obfuscations.

Anderson was the first to demonstrate a *Section Create* action, filling the section contents with random bytes and using random names for the new section table entries whereas our action targets specific section names and virtual sizes using the Shapley results. We follow the convention of prior authors in calling our action novel due to the novel targeting mechanism as opposed to the targeting of some new structure

Kreuk et al. (2018); Chen et al. (2019). The raw and virtual sizes Anderson used are simply the file-aligned and section-aligned size of the random data, respectively. The raw and virtual offsets are calculated using the section table entries. This action relied on LIEF version 0.7. A *Section Create* action works by adding a new entry to the end of the section table which resides at the end of the PE headers. A section table entry has the following characteristics: a name (ignored by the program loader), a raw offset (where the section sits in file), a raw size (file-aligned), a virtual address (where the section will be mapped into a process relative to the base address), a virtual size (section-aligned), and memory protections (specify read/write/execute permissions). This entry provides all the information the program loader will need to find the section and load it into into a process.

Since memory is allocated in pages, the *raw size* and *virtual size* are both aligned. Alignment refers to padding a region so that the subsequent region starts at the next memory page. This is especially critical for the *virtual address* as the program loader must specify the memory protections when allocating pages for the process. Sections whose virtual addresses overlap but have different memory protections will cause a critical failure to execute as pages can not be allocated with conflicting protection. Additionally, virtual addresses for sections must be contiguous. If a gap in the virtual space is introduced, then the program loader has no information on what protection to apply to this gap; we learned through testing that this will cause a critical failure. Testing further revealed that the *size of initialized data*, *size of uninitialized data*, and *size of code* fields from the *optional header* do not need to be updated in order to preserve functionality.

Although padding space is typically present, there is no guarantee of this. Additionally, RL agents are typically given a budget of actions, which means several section may be added to a PE file. Ergo, even if 40 bytes (the size of a section table entry) are initially available, successive applications of the *Section Create* action may consume the available padding. If the padding space is consumed, then the *size of header field* needs to be increased to the next file-aligned value. The entire rest of the

file needs to be shifted to a higher offset to prevent code/data from being overwritten. Since the raw contents will be relocated, the *section table* must be updated so the program loader can still find all of the section contents. Each section table entry's raw offset may need to be increased by the same file-aligned amount as the *size of header* field. The *data directories* do not need to be updated since they use virtual addresses to find each section that contains relevant data. Our implementation and the LIEF implementation used by Anderson both perform this shift when necessary.

Song and Kozák point out that the LIEF implementation used by Anderson damages a high percentage of malware. Song found that 63% of malware altered by Anderson's `gym-malware` code had lost functionality, while Kozák found that *Section Create* specifically damaged 96% of altered files. We found that LIEF 0.7 (used by gym-malware) damaged 77% of malware while LIEF 0.9 (used by version 2 of EMBER's feature extraction code) damaged 92% of our functionality test set (using best of three runs with similarity > 95) when we used them to create a section. Examining the LIEF code shows some of the causes.

First, LIEF assumes there is no data in the overlay, which is not a valid assumption even on benign files (e.g., self-extracting Zips). LIEF calculates the raw offset where the new section contents will be placed by looping though the section table. Each entry's *raw size* is added to its *raw offset* and the largest resulting value is file-aligned. This is used as the new section's *raw offset*. This is the same region defined as the start of the overlay, which means that any overlay data will be overwritten. Of our test set of malware files, 23% had data hidden in the overlay.

A second assumption LIEF makes is that no data is hidden in the header padding space (i.e., after the last section table entry). This is false for three of our files. Packers such as UPX and Dwing appear to place pseudo-headers in this region that contain information like the UPX version or virtual address of certain functions within the file. Song was the first to point out that data may be hidden in the header padding Song et al. (2021), but their solution is to simply skip such files (https://github.com/weisong-ucr/MAB-malware/blob/main/arm.py). If nonzero

data is found between the last section table entry and the start of the first section, then the file is merely copied without any section table entry being added. We argue this does not constitute edge-case handling and would actually degrade the performance of the RL agent. This degradation would result from the agent learning that *Section Create* is ineffective without having any way of knowing that the *Section Create* action was not actually executed.

Nevertheless, we tested Song's code and found that it preserved functionality for 79% of malware. We treated files skipped as files failed, but report that the functionality preservation rate would be 84% if we treated them as successfully ignored. This is the highest functionality preservation rate we found in prior work by a large margin and we salute Song's contribution to the space. We do note, however, that Song's code sets all permissions for the added section (Read, Write, and eXecute), which increases rather than decreases evasion against an EMBER GBDT. RWX memory is commonly used by malware authors to rewrite a file after it has been loaded into a process. Since the additional data is not intended for execution, there is no need to set permissions other than read.

Third, when we calculate the virtual address for the additional section we find the largest sum resulting from adding each section's virtual size with its virtual address. This value is then section-aligned and used as the new section's virtual address. An examination of unaltered PE files shows that this this distance matches the distances used in existing PE files and in Song's code. Anderson's `gym-malware` however, sums the virtual address of each section with its raw size not its virtual size. This can result in overlapping virtual pages, which are clearly visible when examining the file's section table entries using `pefile`. This will break functionality if the permissions on the overlapping pages differ or if values from the section with a lower virtual address are overwritten in the process space. Note, LIEF can calculate the new virtual address correctly, but is overwritten by Anderson when calling the `add_section` function. This problem can be corrected by calling `build_imports` before calling `build_binary`. We found that in 33% of files altered by LIEF the new section overlapped with an

existing section unless `build_imports` was called before building the binary, which only occurs in Anderson's code if the *Add Import* action is also chosen on the same binary. Note that even when we call `build_imports` we still see the functionality preservation rate max out at 8% so it is not solely responsible for Anderson's low functionality-preservation rate.

Demetrio published a method for inserting gradient-optimized bytes in the DOS header and DOS stub. This action would overwrite hidden data in the DOS header and DOS stub, a situation we encountered in our dataset. Demetrio also published a variant, *Gradient Extend DOS*, that relocates the *NT header* to a higher offset to make more room for inserting bytes before the *NT header*. Note, Demetrio's action does not add a section table entry, but is similar in concept to one of our action variants that also pushes the *NT header* to a higher offset. We tested Demetrio's method using their code (https://github.com/pralab/secml_malware) and found only 54% of files altered using their *Gradient Extend DOS* method still functioned despite the fact that this method—which does not attempt to add a section—is less invasive than ours. We attribute this to B) our detection of DOS malformation, B) our handling of section table entries with a raw offset of zero and C) our increasing the *size of image* field when we increase the *size of header* field.

First, we encountered a particularly difficult-to-instrument file, referred to henceforth as BEE67 due to its hashed value, that was packed using Dwing. We know this because the phrase "UPackByDwing" is found in the DOS header, where some optional values should be. The DOS header also contains the name of the `KERNEL32.DLL` file and `LoadLibraryA` & `GetProcAddress` functions from that DLL file. This DLL is used to find imported functions in external DLLs. Since Dwing is known to mangle normal control flow to stymie reverse engineering, we believe the function names were placed here to make them available without using the import tables. Since we do not know exactly where/how Dwing makes use of this data, we cannot disturb it. Data is also present in the padding space at the end of the section table, preventing us from naively adding a new section table entry. Examining the

file in Ghidra, we see that this data appears to point to the virtual addresses of a small number of functions contained with the sections. They appear to play a role in Dwing's unusual control flow. We cannot speculate further, but point out that this file represents the most difficult-to-instrument sample in our test set since it has hidden data before and after the relevant areas. Testing showed the file is one of several fatally damaged by Demetrio's action.

Returning to the possible causes of loss-of-function in files altered via Demetrio's method, we find 10 files with sections that have a raw offset of zero prior to instrumentation. These appear to be associated with uninitialized data (e.g., .bss sections). Testing action variants showed that increasing their raw offset can result in loss of function, so unlike Demetrio we do not increase the raw offset of these sections when pushing content to a higher offset. We speculate that section with a raw offset of zero are treated differently than sections with a nonzero raw offset that actually points to raw content on disk.

Third, Demetrio does not increase the *size of image* field when increasing the *size of header* field. Although the *size of image* is a section-aligned value and the *size of header* is a file-aligned value, we found it necessary to increase the *size of image* when pushing the *NT header* to a higher offset in some files. We do not speculate further why this might be the case, merely reporting that it enhanced our functionality preservation rate.

Kozák's edge-case handling for their *Section Create* action consists of A) checking if the padding space is large enough to add a new section table entry and B) relocating overlay data so it is not overwritten. First, Anderson's code relies on LIEF 0.7 which also enlarges the header if necessary by updating each section table entry's raw offset and increasing the size of header and size of image fields. Kozák does not mention checking for code hidden in the padding area. In our method we account for overlay data by using the file-aligned size of the file as the *raw offset* of the new section rather using the section table. Thus, the new section's content can not overwrite any existing data. We experimented with relocating overlay data as Kozák did, but found it did

not improve the functionality preservation rate. In other words, the malware was still able to locate any overlay data it needed when our data was added beyond it. We theorize that malware which references overlay data does not assume the overlay data is beyond the last section, perhaps finding overlay data using an alternative calculation. This value may be hardcoded or calculated using a specific section's raw offset.

From this testing we see that while some prior authors have implemented edge case handling, these edge cases are not sufficient to reliably instrument malware that contains obfuscations or file format deviations beyond overlay data.

5.3.4 Novel Action Variants to Address Obfuscation

To address data hidden in the padding space at the end of the section table, we developed the *Overlap Section Create* action. Inspired by Ange Albertini's proof-of-concept tiny PE file (https://code.google.com/archive/p/corkami/wikis/PE.wiki), our *Overlap Section Create* action overlaps the *DOS header* and the *optional header* in order to create room for a new section table entry even if there is data hidden in the padding space at the end of the section table. In other words, since we cannot write forward, we move the whole *NT header* backware (to a lower offset) to create the necessary 40 byte gap without touching any hidden data that immediately followed the last section table entry. This room is created by overwriting non-essential bytes earlier in the PE headers. Specifically, we overwrite all of the *DOS stub* and some of the *DOS header*. Demetrio argued that, with the exception of the *DOS signature* and *e_lfanew pointer*, these components are optional and can be safely overwritten. When parsing PE files, the program loader checks for the *DOS signature* (which must be at offset $0x0$), then uses the *e_lfanew pointer* (which must be at offset $0x3c$) to find the start of the *NT headers*, which include the *PE signature*, *COFF header*, and *optional header*. This pointer can be changed to any address that fits within four bytes, provided the entire *NT* header is placed at this address without overwriting

any necessary bytes. We set the *time-date stamp* field of the *optional header* to the value $0x34$ knowing that when we relocate the *NT header* so it starts at $0x34$ the *time date stamp* will overwrite the *e_lfanew pointer* with the new start location of the *NT header*, $0x34$. The amount of space this action creates depends on the length of the *DOS stub*, which is not fixed as the Microsoft specification for PE files allow custom *DOS* stubs. Our action checks if the *Overlap Section Create* will create sufficient room to write a new section table entry before using it. In our functionality testing subset, there are three files with data hidden in the header padding area (i.e., after the last section table entry). Two of these are successfully instrumented by *Overlap Section Create* with zero loss of function. One of them, however, has an abnormal *DOS header* and *DOS stub*. This is the Dwing packed file mentioned earlier. In this file, there is no *DOS stub*; instead, the *NT header* immediately follows the *DOS header*. This file demonstrates that Demetrio's assertion that the DOS stub and header can be safely overwritten is invalid on malware files. Since our goal is to create adversarial samples which must—by definition—function, we need to address this.

The *Leap Section Create* action works in the opposite direction from the *Overlap Section Create* by setting the *e_lfanew pointer* to the current start of contents, i.e., moving the *NT header* to a higher rather than lower offset. The start of contents is where the header padding ends and the first section's contents begin. We use this location since it is likely to be a low value. This action is similar to Demetrio's *Gradient Extend DOS* action in that it pushes the *NT header* to a higher offset. However, our action adds a section table entry rather than inserting bytes before the *NT header*. Additionally, our action detects when the DOS header is not safe to overwrite, improving the functionality rate. As mentioned earlier, functionality is also improved by increasing the *size of image* field in addition to the *size of header* field. Although we have not seen this, it is possible, in theory, that a malware file could have a highly anomalous header that of such large size that the pointing past it would require more bytes than are available in the textite_lfanew field. Addressing this is left to future work.

Regardless, we move the *NT headers* to the start of contents location, pushing all section content to a higher offset. The new offset for the section content is calculated by file-aligning the the size of the *NT headers* plus the 40 bytes necessary for a new section table entry. This amount is added to the *size of headers* field and used to update the *raw offset* for each existing section table entry (unless the raw offset of the section is zero, as previously mentioned). The *size of image* field in the *optional header* must also be increased, although in this case we increase by the section-aligned sum of the size of the *NT header* and 40 additional bytes to account for the larger header. The *number of sections* must be incremented. At this point, we have relocated the section contents, moved the *NT header*, and updated all the necessary fields. We also zero out the previous location of the *NT header* fields, but could write adversarial bytes to this space if so desired. We do not overwrite the hidden data, though, since our whole objective was to leave these bytes untouched in case their location is hard-coded and referenced later by the malware file.

Our initial version of this action still broke functionality (0/391 activity detected) for the Dwing file it was designed to fix. In that version, we naively preserved the distance from the hidden data in the header padding to the end of the section table, assuming this distance needed to be maintained. When our action file-aligned the enlarged header, however, it increased the distance between the hidden data and the start of the first section (i.e., start of contents). Through testing we found it was actually this distance from the hidden data to the start of contents that mattered. We therefore modified our action to place any hidden data such that the distance to the new start of code is preserved, which has the additional benefit of making more room for new section table entries. This final modification allows the Dwing file to function even when instrumented.

By function we mean it exhibited 387/391 or 99% of the original activities. This is above the 95% threshold set by Kozák Kozák et al. (2024) and well above the 80% threshold set by Song Song et al. (2020). This delta was consistent across three executions, so it appears deterministic. Examining the file's behavior report in

CAPE, we see that the difference is due to four files not being modified. The names of the files are `ie_to_edge_stub.exe`, `identity_helper.exe`, `msedge_proxy.exe`, `pwahelper.exe`. The use of Microsoft's progressive web application functionality may indicate that the malware is sending/receiving information or the next stage of an attack. Although we cannot guarantee the malware is undamaged, we speculate that the malware is detecting the alteration via checksum or some other metric and opting not to drop these extra files. CAPE indicates that the file does establish persistence by adding itself to Window's `autorun` list and exists soon after checking the time, a classic method for determining if the malware is in a sandbox. This behavior, as well as the numerous files and registry keys altered by the file, indicates that the file is loaded successfully. Over our development period we noticed that when the structure of a binary file is incorrect due to instrumentation, the failure is immediate and complete (0 dynamic artifacts) as the program loader fails to find the required resources or is instructed to create a contradictory structure. Until we properly adjusted the location of the data hidden after the *section table*, the reported activity level for this file was zero. Since the altered file, with our final tweaks, was able to execute a myriad of activities, we know it was loaded into a process without a fatal error.

We combine all four variants of the *Section Create* action into one intelligent method that only escalates to the more complex variants *Overlap* and *Leap* when necessary. The other two variants are simply the *Naive* action which writes the new entry at the end of the section table and the *Push* variant which has been implemented by previous authors. *Push* is called when the padding space is insufficient while *Naive* is called if there is sufficient padding and no data is found in the padding area. Our final action combines all of the previous variants; the pseudo-code is presented below. The action attempts the least invasive changes before escalating to more complex variants if needed. We also tested forcing the actions, i.e., applying the variants without first checking if they are needed or may overwrite data. These full results are related in Table 5.1. This shows that detecting obfuscations and focusing on the least invasive variant plays an important role in our final action's functionality rate.

```
If padding_length >= 40 and sum(padding_bytes[0:40])==0:
    write(section_table_entry)
elif padding_length < 40 and sum(padding_bytes[0:40])==0:
    push()
    write(section_table_entry)
elif sum(padding_bytes[0:40])!=0 and e_lfanew >= 0x80:
    overlap()
    write(section_table_entry)
else:
    leap()
    write(section_table_entry)
```

All of these variants append the section contents to the overlay. However, Gibert and others have noted that overlay data is readily detectable Gibert et al. (2023). Therefore, we produced a variant that places the new data at the start of contents, pushing all other content to a higher offset. This may result in two pushes occurring when *Push* or *Leap* variants are called. Both situations are handled by our code. Placing the new data at this location and creating a section table entry for it prevents heuristic detection of the new data and increases the likelihood that model with a maximum raw input (e.g., MalConv) consumes the new data rather than truncating the file before reaching the new data. This variation, which we dub *Start Section Virtual Negation*, preserves functionality for 95% of files while the standard *Section Virtual Negation* action we created preserves functionality for 97% of malware. By way of contrast, we found that a simple *Overlay Append* action which only adds data to the end of each file had an functionality preservation rate of 98%.

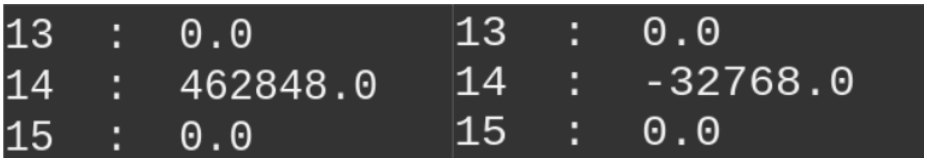


Figure 5.3: On the left we see the feature-space representation of an original malware file. It has a positive value at index 14 which represents the accumulated virtual sizes of all sections whose names hash to that index. On the right we see the feature-space representation of the same malware file once it has been altered by our *Section Virtual Negation* 14 action that adds a new section to the file. The intentional section-name collisions of the "hashing trick" used by EMBER to map the variable sections into a fixed-length feature vector allow us to add colliding sections. This means we can dramatically alter the feature representation at a targeted index be adding a new section whose name will trigger a sign flip when the addition section's virtual size is added to the existing value at that index. We used our malware database and Scikit-Learn's Murmurhash3 algorithm to build a mapping from section names to their signed feature index. Next, we used Thompson sampling (verified via fuzzing) to find the optimal virtual size to allocate for this new section.

Table 5.1: Functionality Preservation Rate for our action variants and most closely related prior actions. The variable action checks which variant to use based on malformation detection, escalating to more complex variants only when necessary (from among Naive, Push, Overlap, and Leap). The SoC variant places the new data at the start of contents (first section offset) rather than the end of the file, pushing all existing contents to a higher offset and updating section table entries. This avoids disrupting the overlay, increases the likelihood the new data will be ingested by raw byte models, and avoids heuristic detection methods.

Action Variant	Functionality Preservation Rate
Anderson: Random Section Create	8%
Demetrio: Gradient Extend DOS	54%
Song: Random Section Create	79%
Koch: Naive Section Virtual Negation	93%
Koch: Push Section Virtual Negation	93%
Koch: Overlap Section Virtual Negation	92%
Koch: Leap Section Virtual Negation	80%
Koch: Variable Section Virtual Negation	97%
Koch: SoC Variable Section Virtual Negation	97%

5.4 Novel Action Effect on Evasion Rate

At this point we have verified that our action, *Section Virtual Negation*, preserves functionality to a higher degree than prior efforts. We have also selected our section names, virtual size, raw size, raw content, and permissions using fuzzing and Thompson sampling. Now, we evaluate the action's evasion rate when used in isolation rather than applied as one of five actions, which is the action budget specified by Anderson. Applied once to each malware file, this action allowed 16% of correctly detected malware files to cross the decision boundary of the EMBER GBDT. A summary of the features affected by this action is provided in Figure 5.4.

We then began testing with the RL agent. We used the DQN agent with 11 Anderson actions (https://github.com/endgameinc/gym-malware) to establish a baseline on our dataset of malware, which consists of Windows PE malware culled from Virushare. We trained the agent for 50,000 rounds (same as Anderson). We then evaluated the agent, testing soft and hard-label performance compared to a random agent. The results are contained in Table 5.2. The soft-label version has access to the score applied by the EMBER target model while the hard-label version only has access to the final label applied (benign or malicious). We can see that the agent is struggling to learn in the hard-label scenario and is outperformed by the random agent, while in the soft-label scenario the agent ties with the random agent. This is in keeping with Anderson's results where the Rl agent narrowly outperformed the random agent on Anderson's data. We evaluated the random agent three times and report the average evasion rate.

Next, we incorporated our four variations (each targeting a different section index) of the novel *Section Virtual Negation* action and retrained the agents. All three agents (random, hard-label, soft-label) report a higher evasion rate. Again, we report the average of three random tests for the random agent. We see that the hard-label agent is still outperformed by random action selection, but that the soft-label agent now outperforms the random agent. Although we were tempted to run the test with longer

Table 5.2: Performance of RL Agents with Stock Actions vs Enhanced Actions

Action Set	Guidance	Evasion Rate
Stock Actions	Random	20%
Enhanced Actions	Random	31%
Stock Actions	Hard-label RL	16%
Enhanced Actions	Hard-label RL	23%
Stock Actions	Soft-label RL	20%
Enhanced Actions	Soft-label RL	36%

Table 5.3: Novel action use in successful (evaded detection) malware files altered by the RL agent. The agent has a budget of up to five action per file. The index targeted by each variant is provided. The last column denotes successful evasion using only novel action variants.

Agent	Index 37	Index 19	Index 22	Only Novel
Hard Label	6/25	1/25	0/25	1/25
Soft Label	23/37	0/37	19/37	12/37

```
Per Dimension Distance:  544
Per Feature Distance:  12
256  hits in  Byte Histogram
183  hits in  Byte-Entropy 2D Histogram
1  hits in  Number of Strings
1  hits in  Average String Length
1  hits in  Number of Characters
95  hits in  Character Histogram
1  hits in  Character Entropy
1  hits in  File Size
1  hits in  Virtual Size
1  hits in  Number of Sections
1  hits in  Section Size Hashes
1  hits in  Section Virtual Size Hashes
```

Figure 5.4: *Section Virtual Negation* Action: We can see the effect the action has on the EMBER features of a malware file. We can see that the virtual size at a single index (19) has been changed and that a host of other features have been affected. For example, the *file size*and *number of sections* have increased. The normalized byte histogram and byte entropy histogram have also changed across several indices since we have appended a number of bytes.

training time or tune the parameters to improve hard-label performance, our initial hypothesis regarded improvements to the available actions alone. From the results we can see that agents improve across the board when given access to more effective actions. Interestingly, we see that a random agent with access to more effective actions is more successful than a soft-label RL agent with more limited actions. Therefore, our hypothesis that evasion rate could be increased via more effective static binary instrumentation methods has been confirmed.

We verified that our action appears regularly in files that successfully evade detection by examining the trajectories (record of actions applied) of each file in Table 5.3. The hard-label agent produced 25 evasive samples. Our action variants appears in 7 of them. One example evaded detection using *Section Virtual Negation* (index 37) alone. The index 37 used most often (6/7) while index 19 is used once. We can see that the soft-label agent, which has access to more useful information in the form of the target model's score, uses our action variants more often: 36/37 successfully evasive samples. Index 37 appears in 23 samples while index 22 appears in 19 samples. In 9 samples both index variants are used. For 12 of the evasive samples only novel action variants were used to evade detection.

Therefore, our primary assertion that increasing the diversity of effective actions available to an RL agent would improve the evasion rate of the agent when all other variables are held constant is correct. However, we now wanted to determine if the guidance mechanism we used—Shapley values on a subset of features chosen via information gain—was a reliable indicator of significance. The goal here was to produce a recommendation to either use the same methodology or find a more effective method.

5.5 Ablation Study

Prior research tends to report only the evasion rate of the agent and, in some cases, which actions were most chosen by the agent. This does not allow us to determine

how valuable an action is in isolation, which would be useful for prioritizing the development of novel actions. Our ablation study—in addition to validating the use of Shapley—provides information on which attributes improved the evasion rate and which did not. The goal is to provide future researchers with insight into which methods are more promising to pursue.

For this ablation study, we first created action variants which used section names that would match to the opposite of the desired sign. We found that these actions yielded similar evasion rates, whereas Shapley indicated these actions would make the files seem more malicious. We also tested with section names not shown to be significant in either direction and found that some of them yielded comparable results to the significant section names. We also tested if th virtual size found during fuzzing was not large enough, adding sections with a virtual size up to 1GB, but found this did not improve the results.

It appears that the Shapley values, at least those obtained using our methodology, do not reliably point to the most significant features. Possible causes for this include our feature reduction using information gain, treatment of each index as a separate feature rather than grouping them together, or Shapley's assumption of feature independence being contradicted. Although this was not the desired result, it did provide a starting point from which to conduct the ablation study, which yielded useful observations regarding the effect of particular features on the EMBER GBDT.

Adding a section affects a number of features. Within the category of section features, we have the name, raw size, virtual size, entropy, and permissions/characteristics. Other features affected include the total raw size, total virtual size, number of sections, byte histogram, byte entropy histogram and a number of string features (string features are affected if the appended content is interpretable as a string). We know that adding a section improves the evasion rate, but that the virtual size/name hashes are not as significant to the overall label as predicted. Therefore, some other feature or set of features must be driving the shift in label. If we could winnow this down, we could find an effective perturbation that targets this subset of features. We

created a set of actions that focus on eliminating as many confounding features as possible while still yielding valid adversarial samples. We maintained our criteria of producing functional binaries rather than feature-space perturbations since we did not consider it productive to find evasions that can only be produced by a set of features that are incompatible with functionality preservation, i.e., a PE with no executable sections.

The total virtual size is calculated by summing the section table virtual sizes. We isolated this action as much as possible by increasing the virtual size of the section with the highest virtual address using the same amount of virtual space as out *Section Virtual Negation* action. If the size of image field is increased by the same section aligned value, functionality can be preserved. This variant would affect that section's hashed virtual size and the file's total virtual size as well a making a very small change to the byte histogram (which happens any time any byte is changed in the file). This did not allow any files to cross the decision boundary, indicating that total virtual size is not significant on its own.

As for the raw size of the file, any large increase in the file size means a significant change is made to the byte histogram and byte entropy histogram. This feature can only be tested in conjunction with the two histograms, at least where realistic input-space perturbations are concerned. In fact, of all the EMBER features, the byte histogram is the most sensitive in that any change made to the file will alter the byte histogram. The byte entropy histogram may also change, although binning the entropy means it is less sensitive to minute changes. Since the byte histogram changes every time the smallest perturbation is made, it is the most sensitive feature. This made us wonder if the GBDT could be paying more attention to it than the other features. To test this theory, we focused on optimizing the content added to a binary with a series of *Overlay Append* and *Section Create* actions.

As mentioned in Section 5.2.3, we already fuzzed singular values for the raw content, finding that 32 was the most effective single byte to repeatedly append. We therefore explored more intelligent methods for choosing which bytes to append. The first we

checked was based on Chen's *GradCAM Benign Overlay Append* action Chen et al. (2019). This action uses MalConv as a surrogate in combination with the GradCAM explainability metric. We fed a trained MalConv model a large number of benign files, recording the file offsets of significant (according to GradCAM) chunks within these files. We then carved out these chunks and appended them to the overlay of benign files. Appending 6MB of benign data to the overlay allows 45% of files to cross the decision boundary, which is much higher than our best Section Virtual Negation action. However, appending 6MB of random data to the overlay allows 47% files to cross the decision boundary, narrowly outperforming the more (apparently) intelligent method. We tested adding different amounts of random data (1-8 MB), finding that evasion plateaued at around 4MB of random data with 63% crossing the decision boundary. GradCAM benign data peaked at 5MB with 47 crossing the decision boundary. Therefore, the amount of content to add, which affects the raw size feature, is dependent on the type of content. The relatively high evasion rate of *Random Overlay Append* seemed counter to intuition since high entropy is associated with packed data, an obfuscation commonly although not exclusively associated with malware. We therefore continued testing action variants that add random data.

To test whether location of the random data is at all significant, we compared appending data in the overlay (*Random Overlay Append*) to placing the data earlier in the file. This requires shifting existing content to a higher offset and updating the raw offset field of each section table entry for each section. The new offsets must be file-aligned values. We term this action *Random Shift Section* to denote that it is similar to Demetrio's *Gradient Shift Section* action. Note, the data is not placed in a section so functionally this is the same as placing the data in the overlay. Appending 2MB of random data using *Random Overlay Append* allows 22% files to cross the decision boundary, while *Random Shift Section* allows 67% to cross. However, if we append 4MB using *Random Overlay Append*, we see that 63 cross the boundary compared to 71 for *Random Shift Section* with the same amount of data. Why could inserting data at an earlier location result in higher evasion rates when 2MB of random

data are added but show marginally different results at $4MB$ of added data? The raw offset of sections are not recorded in the EMBER features and, as mentioned previously, the byte histogram and byte entropy histogram are location agnostic.

However, the EMBER feature extraction code shows that the entropy of each chunk is calculated sequentially then binned. By inserting data at an earlier location and pushing existing contents downward, we change which bytes are within the boundary of each chunk, which alters the entropy score applied to each chunk. We posit that as we add more and more random data, this shift becomes less significant due to the huge amount of high entropy data being added. We can confirm this by examining the byte-entropy histograms of three randomly-chosen altered files. As our metric we sum the absolute value of the difference in byte-entropy at each index of the histogram. This allows us to measure how different two byte-entropy histograms are. As a baseline we compare three pairs of files altered by *Random Overlay Append* with 2MB of data. The distances are .0030, .0021, and .0035. This tell us how much the entropy may differ when the same method is applied to these three files due to random variation. Note, the entropy values at each index are quite small, yielding very small difference values. If we compare these three files with their *Random Section Shift* equivalents, then the distances increase to .0952,.0772, and .1125—an order of magnitude shift in each file. In other words, inserting data earlier in the file affects the byte entropy more than we would expect considering the same amount of random data is added. However, the byte-entropy distance between *Random Shift Section* and *Random Overlay Append* files decreases to .0074, .0059, and .0101 when 4MB of data is added, which is the same order of magnitude difference we expect from running the same random action twice. In other words, as the data increases the difference in entropy between the two methods decreases. This could explain why *Random Shift Section* is more effective than *Random Overlay Append* with smaller amounts of data but is only marginally more effective than *Random Overlay Append* as the amount of appended data increases. Thus location has a weak but nonzero effect on evasion rate.

We next tested adding the random data in a section (*Random Section Create*) rather than leaving it in the overlay (*Random Overlay Append*). The evasion rate drops from 63% to 37% when the section is marked read-only and 8% when the section is marked executable. This is in contrast to the results from Chapter 3 where adding data in a section marked executable was markedly more successful at evading COTS tools than adding data to the overlay. Since this data is random rather than pulled from benign files or optimized via the gradient, the fact that *Random Overlay Append* allowed a large number of files to evade detection seemed curious at first. The malware classification problem is a binary one; therefore, this high evasion rate implies that—at least for an EMBER GBDT trained on the EMBER data—there is some association between a relatively large amount of random data that is outside of a section and the benign label. Note, random data has high entropy which makes it difficult to distinguish from highly compressed data. Although data outside of a section deviates from the format specification, there is one benign use-case for this arrangement: self-extracting Zip files.

As mentioned in Chapter 2, self-extracting Zip files are actually a PE+Zip polyglot where the Zip is placed in the overlay of the PE file. Since the Zip contents are compressed, this would result in regions of high entropy outside of an existing section. Self-extracting Zips are commonly used as installers for software, which makes them readily available when scraping the web for benign PE files. If these files are present in the EMBER dataset, they would explain the association between random data outside of a section and the benign label. Since the benign EMBER files were not released due to copyright restrictions, we do not have direct access to these files. We have contacted the EMBER authors and provided them with a script that recurses through a directory executing `binwalk` on each file to find any "archive" hits in the `binwalk` output. Their initial response was willing, but we have not heard back yet with results from running the script. In the meanwhile we analyzed one of our own polyglot data sets using this script. Although we had already taken steps to find and remove installer polyglots during that project, this script was able to find a single

remaining example of a self-extracting Zip in our PE data. If the EMBER authors are able to confirm our suspicion, this insight might yield a recommendation to treat self-extracting Zips as separate from PE files so as to avoid accidentally introducing a vulnerability to the *Random Overlay Append* action.

5.5.1 Bound PGD Section Create at SoC Action

The ablation study of our novel *Section Virtual Negation* action revealed that the Shapley feature importance metrics were not accurate. However, this study did reveal that byte content, as represented by the byte histogram and byte entropy histogram is highly significant. By ignoring the section features completely, we can reach an evasion rate of 71% with our *Random Shift Section* action. Although learning that EMBER is vulnerable to byte contents despite extracting a large number of features from the header and section table is valuable in its own right, the action that—thus far—maximized evasion is not particularly novel or interesting. We therefore set about creating a more effective action using the insight we'd gained. Our new approach targets the same feature–the byte histogram–but uses projected gradient descent on a surrogate model. The surrogate is a neural network trained on the EMBER features set. We had initially avoided the surrogate+gradient route because creating a reconstruction algorithm for the entire EMBER feature set, while useful, is a major undertaking that would require many simplifying assumptions and careful heuristics to prevent the resulting combination of input-space changes from damaging functionality. Thanks to our ablation study, however, we now know that we can focus on the byte histogram. Creating a tractable reconstruction method that does not interfere with functionality is straightforward for this feature.

Therefore, we began by training a differentiable surrogate model on the EMBER features. By training a convolutional neural network on the same features as the target model, we are able to rely on the transfer property. As prior researchers have shown,

attacks trained against a MalConv surrogate do not transfer well to an EMBER GBDT due to differences in the feature space Demetrio et al. (2021c).

We used the adversarial robustness toolkit (https://github.com/Trusted-AI/adversarial-robustness-toolbox) to conduct a projected gradient descent (PGD) attack on the surrogate model. Unlike prior approaches, we masked the attack vecftor so that only the byte histogram was allowed to change. Further, we used each input file's byte histogram to generate a minimum bound for each index in the byte histogram. This forced the attack to find additive changes to the histogram. Since the histogram is normalized, reduction attacks are also possible, but left to future work. By comparing the masked and bound vector generated by the PGD attack to the original file's byte histogram, we know which indices need to increase by what proportion to move the original file toward the optimized adversarial representation.

We then produced our own own reconstruction method to map the necessary changes back to the input space. Our method adds N bytes to the file, calculating how many bytes of each index (hex value) to add to the file such that the resulting file is drawn close to the optimal vector as generated via PGD. We make no guarantee how close this resulting vector is to the optimal vector. Since this action appends bytes to the overlay using PGD, we name it *Bound PGD Overlay Append* and set an append budget of 4MB. This action, applied in isolation from any other guidance mechanisms or actions, allows 93% of altered files to bypass the EMBER GBDT. This evasion rate is much higher than that achieved by the RL agent or the random append actions.

A search of existing literature found a recently released paper that also targeted the byte histogram Gibert et al. (2023). We do note, however, that this paper used a generative adversarial network to generate an optimal vector of bytes which were then appended to the file, i.e., a *GAN Overlay Append* action. Against EMBER, this action yielded an evasion rate of 17% Gibert et al. (2023). Recall that a GAN uses a generator and discriminator trained in unison. Although these models are useful, they are notorious for getting stuck in performance plateaus if one of the two models improves more rapidly than the other. Since our method uses the gradient of

the surrogate model without needing to train a generator and discriminator, it does not have this potential issue. We note that we used different data sets so a direct comparison should be taken with a grain of salt. The difference in performance may be due to implementation differences in our reconstruction method, a plateau issue during training of the GAN, or stark differences in the malware dataset. We also note that our version adds 4MB of data whereas Gibert used up to 56MB of additional data Gibert et al. (2023).

Gibert notes that one glaring weakness of their approach is that overlay data can be readily detected via a simple heuristic. If any data exists beyond the highest file-aligned value found by summing each section's raw size with its raw address, then overlay data is present. Industry tools—e.g., Virustotal—have implemented such heuristics and apply an "overlay" tag to analyzed files if this extra data is found. Unlike Gibert, we can address this issue using knowledge from our prior action development. Instead of placing the data at the end (overlay), we place it at the old *start of code* location, shifting all subsequent content to a higher offset. Furthermore, we add a section table entry to match the new data so it appears original. Note, the checksum could still be used to detect alteration. However, any overlay calculation based on parsing the section table entries will not reveal the new data since it corresponds to a section table entry. By placing the contents earlier in the file we also ensure that models with a maximum capacity for raw input—e.g., MalConv—would ingest this additional content even when their maximum capacity would truncate the input before overlay data is read. In this manner, we make the attack harder to detect and prevent. We named this variant *Bound PGD Section Create Create at SoC* (Start of Contents) and found that it has an even higher evasion rate of 100% against the EMBER GBDT on our small functionality verification dataset and 97% against a larger random sample of 300 files.

This evasion rate is much higher than the best evasion rate of the reinforcement learning agent using our prior action. It further indicates that the surrogate approach may be superior to feature importance via Shapley values, at least when the Shapley

values are calculated using our methodology. Of course, training a surrogate on the same features requires prior knowledge of the feature space. As our survey points out, however, even the RL approach relies on some prior knowledge of the feature space of a target model (static vs dynamic). Since this method was so effective on its own, we did not repeat the integration into the RL agent. That EMBER, for all its header and section table features, can be thoroughly evaded by focusing on the byte histogram is contrary to our expectation.

We did another experiment to determine if we could decouple the byte histogram from the byte entropy histogram by scrambling the appended data. In our stock reconstruction method, each hex value is appended sequentially. In other words, we append the value zero N times, the value one Y times, and so on. This local repetition reduces the entropy of these new chunks of data. If, however, we randomize the append order, then the entropy increases and the evasion rate is reduced. Therefore, although our action is masked and bound such that it targets only the byte histogram, we unwittingly created an action that impacted the byte entropy histogram in a productive manner (by lowering it in the appended chunks). This indicates that both values are significant, although the byte histogram appears to have a stronger effect on evasion rate. This may be due to our targeting of the action as opposed to targeting the byte entropy histogram. Future work might target the byte entropy histogram, but would necessitate a more sophisticated reconstruction method that accounts for local values and their entropy.

Finally, we repeated our functionality verification testing with this novel action, finding that it preserved functionality for 95% of malware. This preservation rate was the same for both the original version which added the section content at the end of the file and the subsequent version which inserted the content earlier in the file.

Our reconstruction algorithm is greedy, appending each byte repeatedly until the necessary increase for that byte is reached. With a budget of $4MB$ this means three bytes are added, most commonly this is $0x0$, $0x31$, $0x90$.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

6.1.1 Summary of Results

We first addressed feature extraction evasion by producing polyglot detection and remediation tools. A deep learning approach was the most effective at classifying files in both the binary and multi-label versions, outperforming commonly used tools for file identification. As Bridges Bridges et al. (2020) demonstrated in an evaluation and we confirmed through our paper Koch et al. (2024), polyglots are effective at evading detection by COTS tools and are in-use by malware authors. Since format identification allows for format-specific feature extraction, it enables detection models to extract only relevant features while discarding noise. To continue to use format-specific detectors it is therefore necessary to filter out polyglots or route them to multiple detectors, which is the service provided by out binary and multi-label detectors. This work resulted in two tech transfers, one workshop paper, and an under-review conference paper. Now that we could be confident that extracted features were representative of the correct file format, we could consider how to manipulate the file so that the extracted features would evade detection despite being valid.

We began our perturbation research by building a set of instrumentation actions which we tested sans guidance against two COTS detectors in use by a sponsor. We found that actions which interacted with the section table of the PE file were the most effective at shifting the label (up 56% evasion rate) applied by both models. This work was recently granted a patent (US 12019746) and licensed to a startup. It also prompted us to investigate the PE file structures to a greater depth and begin a survey of guidance mechanisms that could optimize action selection/content.

The survey showed that instrumentation action development was a major limiting factor in prior research. Since PE files, especially the most common X86/64 architecture, are opaque in nature and malware authors regularly obfuscate control flow, it is not trivial to alter a binary while maintaining the true label. Prior authors that utilized RL algorithms have largely focused on the novel application of different RL algorithms to the alteration problem while ignoring improvements that could come from novel action development. As our results against the COTS detectors demonstrated, novel actions can be effective sans guidance, leading us to wonder if we could advance state of the art by finding a more effective action and tying it into an existing optimizer rather than focusing on selecting/tuning the optimizer.

Of the two most common target models—MalConv and an EMBER GBDT—the GBDT is the more interesting target since it is not differentiable. This means that finding a more evasive action either requires a black-box explainability method or a differential surrogate that leverages the transfer property. In order to develop a more powerful action against the GBDT, we therefore investigated these two approaches.

The survey also revealed that functionality preservation is understudied and that published criteria for validating preservation vary widely, with a number of authors performing no validation at all. By definition an adversarial sample must have the same label as the original sample Szegedy et al. (2013), which Demetrio defines as possessing the same behavior as the original sample for the specific case of adversarial binaries Demetrio et al. (2021a). Since malware is known to deviate from the file format specification to evade detection, functionality preservation can not be assumed.

This was demonstrated by Song and Kozák Song et al. (2020); Kozák et al. (2024) who found that prior work damaged functionality for a high percentage of files. Therefore, we stipulated that a necessary criteria for our novel action was a high degree of functionality preservation using thorough and clear criteria. The survey is currently under review by ACM's Computing Surveys.

Building on results from an explainability project that used Shapley values, we focused on the virtual size of sections. We found that we could flip the sign of the virtual size features by adding new sections with names that exploit the intentional hash collisions used by the EMBER features to bin the section attributes. Using our malware data set, we were able to predict which section names would result in either a positive or negative value being extracted to a particular feature index. By adding a new section that collides with the targeted existing section, we could affect the feature representation without interfering with an existing section. We tested functionality of our *Section Virtual Negation* action, comparing it to the most similar actions in prior work.

We found that prior work largely failed to preserve functionality by making assumptions based on the PE documentation rather than an examination of deviations found in malware files. We were able to increase the functionality preservation rate to 97% for our action by experimenting on packed malware files. Having demonstrated that our action has a much higher evasion rate than the nearest published work, we conducted limited optimization of the section attributes. Using Thompson sampling and fuzzing, we selected the optimal raw size, virtual size, and permissions. We also experimented with the section content, finding that certain bytes were more effective than others when appended repeatedly. We integrated four variants of this action into Anderson's `gym-malware` environment, finding that the novel actions were regularly present in successful trajectories and increased evasion rate for the random, soft-label, and hard-label agents by a noticeable margin.

Having validated our primary hypothesis that we could improve evasion rate by developing a more effective action that preserves functionality to a high degree, we

turned to an ablation study to confirm our development methodology, i.e., the Shapley results. We found the size and sign of the hashed virtual size was not as significant as the Shapley results had indicated. Instead, our ablation study showed that the section content which we had optimized in a limited fashion using fuzzing and Thompson sampling had the largest effect on the evasion rate. This content is represented in the feature space by the byte histogram and byte-entropy histogram. Since optimizing the byte content of even a moderately large section would be computationally expensive via fuzzing or Thompson sampling, we turned to the surrogate approach to find an action that targeted these features. We used a CNN trained on the EMBER features as our differentiable surrogate and conducted a projected gradient descent attack. We masked and bound the adversarial vector so that only additive changes to the byte histogram were allowed. We then developed a reconstruction mechanism to insert bytes so that the altered file was pulled toward the adversarial vector. We found that we could improve the evasion rate to 100% by adding $4MB$ of data, inserting the data at the start of contents rather than the overlay, and adding a section table entry corresponding to the additional content. We checked the literature for a similar method, finding that Gibert had also targeted the byte histogram. However, Gibert used a GAN to directly optimize the bytes appended to the overlay whereas we combined a masked and bound PGD attack on a differentiable surrogate model with our own reconstruction method. Our evasion rate is much higher than that of Gibert (100% vs 17%) against an EMBER GBDT. GANs are well-known to suffer from stability problems during training, possibly explaining some of the difference in evasion rate. Placing the additional content in a section at the start of code rather than dumping it in the overlay is responsible for 7% of the difference in evasion rate. We did limited ablation on this second action, finding that our reconstruction method had a positive effect on the byte-entropy histogram by appending bytes sequentially. This ensured that the added data had the lowest possible entropy. In other words, when we scrambled the append order, the evasion rate decreased markedly. This indicates that the low entropy of our naive reconstruction mechanism was important

to the evasion rate, which may also have been a factor in our evasion rate exceeding that of Gibert. We tested this action for functionality preservation, finding that even though we inserted data at the start of code and altered the section table, we preserved functionality for 95% of altered files.

Perhaps our most significant find was EMBER's vulnerability to the byte histogram. In a way, this means that EMBER features are not more resilient to content-based attacks than raw-byte models. That attacks from prior work (e.g., Demetrio Demetrio et al. (2021c)) did not transfer well from a raw byte surrogate to an EMBER model may be due to the difference in feature extraction method (binning into histogram vs embedding into a high-dimensional space) rather than the additional features EMBER extracts. Despite parsing the section tables and numerous header fields, an EMBER GBDT is largely vulnerable (93%) to an overlay append attack. Adding a section table entry and inserting contents earlier did improve the evasion rate another 7%, but were not the prime contributors to the delta in evasion rate. Ergo, for all its additional complexity, the EMBER features are heavily reliant on a malleable representation of the raw content rather than the structures of the header and section table. Unless a model is incentivized to make full use of available features, we should not assume that a more sophisticated set of features creates a more robust model.

Finally, our ablation study revealed a vulnerability to high entropy data outside of a section (e.g., the overlay). We believe this is evidence of self-extracting Zips in the EMBER dataset, an interesting artifact that may have poisoned the detector by creating an association between the benign label and high-entropy data outside of a section. We sent the EMBER team a script to search for these polyglots in their dataset, and await their response.

6.1.2 Recommendations

Our results provide a number of take-aways related to detection evasion and detection improvement.

6.1.2.1 Evasion

The hunt for highly evasive actions is most efficiently done in the feature space using a differentiable surrogate trained on the same features. The much higher evasion rate yielded by the PGD action is proof of the effectiveness of this route. Shapley has limitations due to assumptions of independence between features Kumar et al. (2020), although our methodology of treating indexes as independent may have contributed to its lack of success.

Training a differentiable surrogate is predicated on knowledge of the feature space, which may not be available. Queries could be used to determine which features are extracted by a black-box target. This fingerprinting-via-evasion would require highly evasive samples be submitted to the target detector since COTS detectors only return hard-labels. We would only know a feature is present if we successfully evaded the model using an attack that targeted that feature. Since COTS tools do not work in the feature space, one would need to submit complete binaries. As mentioned in Chapter 3, a stable of perturbation methods like our *Bound PGD Histogram Section Create* action that target specific feature groups would be needed to create this query system. Once this library of effective actions was assembled, genetic or RL methods could be used to find the optimal combination of actions to evade a particular target. This could be used to develop a fuzzy sense of the features extracted by a target model, which could then be used to develop more efficient evasion attacks against the target model using a customized surrogate.

Functionality preservation needs to be thoroughly tested before and after alteration. Our results show that even without alteration there is a high degree of variability in malware samples. Therefore, we needed to filter out a large percentage of files that showed no or unreliable behavior prior to testing in order to guarantee that the file was functional prior to alteration. Prior claims of functionality preservation without this first step leave prior results open to question since no loss of function will be detected in a file that initially displays no behavior. We are the first to study malware files that

broke after alteration, finding that packers routinely violate file format specification. Therefore we were not only able to explain why functionality preservation assumptions were invalid but also find solutions. We were able to alter malformed by checking for hidden data and manipulating the *e_lfanew* pointer to either overlap or leap the headers. We would strongly recommend that future researchers pay attention to the malware itself, as these files provide not only lessons to be learned from edge-case handling but can also demonstrate successful file format deviations that do not break functionality. In a way, malware authors are an example of crowdsourced fuzzing. By studying their methods for evading detection we can develop simple yet effective instrumentation actions.

Finally, functionality verification requires a sandbox and is a time-consuming process. Emergent research with LLMs as decompilers might yield a method for functionality preserving alterations that can be applied to an intermediate representation (e.g., LLVM or P-code) and mapped back to a functional binary without needing the sandbox for validation. Current results indicate this procedure is in its infancy Tan et al. (2024), but may make rapid progress given current commercial interest in LLMs.

6.1.2.2 Detection

First, detection based on structural features of PE files is not reliable. The format specification for PE files tolerates many possible transformations that affect EMBER features, yielding a large attack surface. Detection systems would do well to restrict themselves to static features that are difficult to transform without loss of function. The control flow graph (CFG) is one example of a feature based on functionality rather than structure. A model trained on the CFG would not be sensitive to structural alterations that have no effect on control, rendering most published methods harmless. The use of location invariant and/or normalized features in the EMBER feature set increases the attack surface by allowing us to alter features without altering input.

In other words, we can change the byte histogram and section virtual sizes without touching the original bytes or sections. This is a problem for robust detection.

Switching to behavioral rather than structural features would not provide a foolproof solution as Sharif demonstrated Sharif et al. (2019). However, editing the CFG without loss of function requires more advanced tools and methods, including conservative disassembly and reconstruction. This raises the minimum threshold of necessary attacker knowledge/capability. As our work indicates, the current threshold is practical machine learning skills (e.g., use of open-source adversarial attack methods, training/testing of open-source models), knowledge of the PE structure (e.g., Microsoft documentation, Ange Albertini and Ero Carrera's open-source projects), and extensive experimentation. Editing of a PE file's executable bytes would at least require the use of a tool like Grammatec's DDisasm, OFRAK, or Sharif's conservative disassembly.

One could argue that dynamic detection is more reliable since it ignores static obfuscations and relies instead on actions taken by the file. However, COTS tools must maintain a high throughput, encouraging the use of static features despite their unreliability. If the time-consuming nature of our functionality-preservation testing was any indication, the expectation that all malware detection will someday be dynamic in nature is naive. Ergo, the search for static yet reliable features will continue despite their current unreliable nature.

If we had to make recommendations regarding structural models, we would first consider feature extraction evasion. Malware detectors need to verify that hidden functionality is not lurking in non-extracted features. Polyglots are currently in use by malware authors and have a demonstrated ability to evade detection by leading COTS tools Koch et al. (2024); Bridges et al. (2020). Otherwise, it is straightforward to hide functionality using a polyglot file. As Chapter 2 demonstrates, format-agnostic models trained on raw bytes can be used to detect polyglots prior to format-specific feature extraction. Filtering out polyglots allows us to focus in on PE specific fields. Since we can now treat the extracted features as representative, we would want to switch tacks, ignoring as many easily-altered features as possible. The issue with the

byte histogram from the EMBER features is that all sense of relative location is lost. The translation invariance of the byte histogram in the EMBER features appears to simplify the perturbation task in much the same way as the global max pooling layer of the MalConv model. In other words, we can add content wherever tolerated by the format and the model will not be aware that that the additional content is not in a normal well-formed location. However, making a model sensitive to location can introduce vulnerability as well. The sequential window used to calculate the byte entropy histogram, for instance, can be abused by shifting content so the window start and stop indices are changed, which results in different entropy values being assigned to each bin. As our work shows, relocation of raw assets is somewhat trivial given knowledge of the format specification. Since the PE format specification uses a number of raw pointers to find assets, we can abuse these to insert data at numerous locations without disrupting section contents. We argue strongly that the byte and byte-entropy histograms should not be used as features. They are too easy to perturb without loss of function. Since the byte histogram is normalized, negative alterations are possible as well as positive ones. In other words, by increasing the file size and occurrence percentage for some files we automatically reduce the occurrence of other bytes, again yielding a large attack surface. N-grams might be an effective method that brings some notion of location relative to other bytes into play without introducing a sensitivity to exact locations. The idea here would be to encode sequences that are more difficult to alter such as the executable sections. Of course more executable sections can always be added up to the max of 96 sections, so evasion is possible for this feature. Virtual addresses are a bit more difficult to alter than raw offsets since the data directory entries must be considered in addition to the virtual offsets of sections. Development of a virtual offset perturbation would be interesting future work; however, our experience thus far indicates that such a method, while not trivially easy to create, would certainly be tractable. Therefore, we do not hold much confidence that a virtual address feature would be significantly more reliable. Thus, we seem to return always to the notion of filtering out any bytes that are not actually part of

the control flow even if they are marked executable, i.e., using the control flow graph. A data flow graph could also be used to determine which data is ever referenced during execution. Anything extraneous could then be discarded. Opaque predicates as mentioned by Sharif mean this process would not be without limitations Sharif et al. (2019), but would be worthwhile simply to reduce attack surface and increase difficulty. While improvements could no doubt be made to structural features, our work thus far indicates the return on investment would not be sufficient to justify the necessary development and testing only to yield a model that is marginally more difficult to attack.

6.2 Future Work

The use of a differentiable surrogate model trained on the same feature set was the most effective attack by a large margin. This approach was possible because a) the EMBER features are open-source and b) our ablation study pointed us toward a subset of features for whom a reconstruction method was straightforward (byte histogram). To extend this approach to a hard-label black-box scenario, initial work would need to be done to determine, as much as possible, the feature set of a target model via queries. This is one area where our approach, as opposed to feature-space perturbations, provides an advantage. We cannot submit altered features to a COTS target model; we can only submit actual binaries. Thus, we can only query the model by submitting complete binaries rather than extracted features. Since COTS tools typically may only output a hard label, we would only be aware of a feature's extraction and significance if altering this feature allowed the altered file to cross the decision boundary—a rather high bar. Malware feature extraction spans a wide variety of possible features from static to dynamic which would each need to be addressed with actions that are evasive and functionality preserving. Rosenberg trained surrogates on single features and found attacks effective against each feature. Although the attacks were less successful against a hybrid feature surrogate, they still transferred to some

degree. Therefore, the most promising angle of attack against a truly unknown target would consist of building a stable of perturbations that have been optimized to target solitary features. These methods would be tested in isolation. If no evasions are found, then testing of combinations would proceed. A genetic algorithm that selects perturbation combinations seems like a natural fit to solve this problem. Before this algorithm could be tested, however, we would need that set of effective perturbations. Our work provides one such perturbation method. The development of more evasive yet functionality preserving perturbations is the direction we would like to proceed with this research in the hopes of eventually building this truly black-box model.

Bibliography

Aghakhani, H., Gritti, F., Mecca, F., Lindorfer, M., Ortolani, S., Balzarotti, D., Vigna, G., and Kruegel, C. (2020). When malware is packin' heat: Limits of machine learning classifiers based on static analysis features. In *Proceedings of the Network and Distributed Systems Security (NDSS) Symposium*. 128, 147

Albertini, A. (2015). Funky file formats. *International Journal of Proof-of-Concept or Get The Fuck Out*. 12, 15, 37, 42

Anderson, H. S., Kharkar, A., Filar, B., Evans, D., and Roth, P. (2018). Learning to evade static PE machine learning malware models via reinforcement learning. *CoRR*, abs/1801.08917. 68, 69, 99, 102, 122, 131, 132, 143, 169

Anderson, H. S., Kharkar, A., Filar, B., and Roth, P. (2017). Evading machine learning malware detection. *Black Hat*, 2017. 99, 122, 130, 144, 169

Anderson, H. S. and Roth, P. (2018a). Ember: an open dataset for training static pe malware machine learning models. *arXiv preprint arXiv:1804.04637*. 9, 52, 100, 104, 109, 116, 121, 132

Anderson, H. S. and Roth, P. (2018b). Ember: An open dataset for training static pe malware machine learning models. *arXiv preprint arXiv:1804.04637*. 13, 14

Aryal, K., Gupta, M., and Abdelsalam, M. (2022). A survey on adversarial attacks for malware analysis. 85

Aslan, O. and Samet, R. (2020). A comprehensive review on malware detection approaches. *IEEE Access*, 8:6249–6271. 93, 94, 95, 96

Bazrafshan, Z., Hashemi, H., Hazrati Fard, S. M., and Hamzeh, A. (2013). A survey on heuristic malware detection techniques. In *IKT 2013 - 2013 5th Conference on Information and Knowledge Technology*, pages 113–120. IEEE. 93, 96

Beebe, N., Liu, L., and Sun, M. (2016). Data type classification: Hierarchical class-to-type modeling. In Peterson, G. and Shenoi, S., editors, *Advances in Digital Forensics XII*, pages 325–343. Springer International Publishing. 19

Beebe, N. L., Maddox, L. A., Liu, L., and Sun, M. (2013a). Sceadan: Using concatenated n-gram vectors for improved file and data type classification. *IEEE Transactions on Information Forensics and Security*, 8(9):1519–1530. 19

Beebe, N. L., Maddox, L. A., Liu, L., and Sun, M. (2013b). Sceadan: Using concatenated n-gram vectors for improved file and data type classification. *IEEE Transactions on Information Forensics and Security*, 8(9):1519–1530. 41, 56

Brandis, R. (2009). Exploring below the surface of thegifar iceberg. 20

Bridge, K. et al. (2021a). Pe format. 89, 90, 155, 166

Bridge, K. et al. (2021b). Pe format. 179, 185

Bridges, R. A., Oesch, S., Verma, M. E., Iannacone, M. D., Huffer, K. M., Jewell, B., Nichols, J. A., Weber, B., Beaver, J. M., Smith, J. M., et al. (2020). Beyond the hype: A real-world evaluation of the impact and cost of machine learning-based malware detection. *arXiv preprint arXiv:2012.09214*. 13, 36, 41, 43, 210, 217

Bulazel, A. and Yener, B. (2017). A survey on automated dynamic malware analysis evasion and counter-evasion: Pc, mobile, and web. In *Proceedings of the 1st Reversing and Offensive-Oriented Trends Symposium*, ROOTS, New York, NY, USA. Association for Computing Machinery. 95

Campion, J. and Laloge', H. (2018). Polydet. 56

Castro, R. L., Schmitt, C., and Dreo, G. (2019a). Aimed: Evolving malware with genetic programming to evade detection. In *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 240–247. 119, 154

Castro, R. L., Schmitt, C., and Rodosek, G. D. (2019b). Armed: How automatic malware modifications can evade static detection? In *2019 5th International Conference on Information Management (ICIM)*, pages 20–27. 101, 119

Chen, B., Ren, Z., Yu, C., Hussain, I., and Liu, J. (2019). Adversarial examples for cnn-based malware detectors. *IEEE Access*, 7:54360–54371. 12, 104, 106, 117, 118, 120, 126, 129, 141, 187, 203

Chen, J., Jiang, J., Li, R., and Dou, Y. (2020). Generating Adversarial Examples for Static PE Malware Detector Based on Deep Reinforcement Learning. *Journal of Physics: Conference Series*, 1575(1):012011. 130, 132, 136, 138, 162, 169

Clunie, D. A. (2019). Dual-personality dicom-tiff for whole slide images: a migration technique for legacy software. *Journal of pathology informatics*, 10. 37

Cohen, F. (1987). Computer viruses: Theory and experiments. *Computers & Security*, 6(1):22–35. 93

Collberg, C., Thomborson, C., and Low, D. (1997). A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand. 122, 139

Coull, S. E. and Gardner, C. (2019). Activation analysis of a byte-based deep neural network for malware classification. 109, 110

Cyber, S. (2019). Cylance, I Kill You. https://skylightcyber.com/2019/07/18/cylance-i-kill-you/. 77

Darwin, I. et al. (2019). file. 17, 38, 56

Demetrio, L., Biggio, B., Lagorio, G., Roli, F., and Armando, A. (2019a). Explaining vulnerabilities of deep learning to adversarial malware binaries. *arXiv preprint arXiv:1901.03583*. 23

Demetrio, L., Biggio, B., Lagorio, G., Roli, F., and Armando, A. (2019b). Explaining vulnerabilities of deep learning to adversarial malware binaries. 115, 127

Demetrio, L., Biggio, B., Lagorio, G., Roli, F., and Armando, A. (2021a). Functionality-preserving black-box optimization of adversarial windows malware. *IEEE Transactions on Information Forensics and Security*, 16:3469–3478. 88, 115, 140, 147, 154, 155, 211

Demetrio, L., Coull, S. E., Biggio, B., Lagorio, G., Armando, A., and Roli, F. (2021b). Adversarial EXEmples. *ACM Transactions on Privacy and Security*, 24(4):1–31. 36

Demetrio, L., Coull, S. E., Biggio, B., Lagorio, G., Armando, A., and Roli, F. (2021c). Adversarial exemples: A survey and experimental evaluation of practical attacks on machine learning for windows malware detection. 24(4):1–31. 2, 85, 90, 100, 103, 104, 105, 110, 115, 116, 117, 126, 127, 128, 154, 165, 173, 207, 214

Desjardins, B., Mirsky, Y., Ortiz, M., Glozman, Z., Tarbox, L., Horn, R., and Horii, S. (2020). Dicom images have been hacked! now what? *American Journal of Roentgenology*, 214:4:727–735. 14, 20, 37

Dhanjani, N., Hardin, B., and Rios, B. (2009). *Hacking: The Next Generation*. O'Reilly Media. 20

Duan, Y., Chen, X., Houthooft, R., Schulman, J., and Abbeel, P. (2016). Benchmarking deep reinforcement learning for continuous control. In *International conference on machine learning*, pages 1329–1338. PMLR. 135

Díaz, V. (2022). Monitoring malware abusing cve-2020-1599. https://blog.virustotal.com/2022/01/monitoring-malware-abusing-cve-2020-1599.html. Accessed: Apr. 25, 2023. 238

Ebrahimi, M., Pacheco, J., Li, W., Hu, J. L., and Chen, H. (2021). Binary Black-Box Attacks Against Static Malware Detectors with Reinforcement Learning in Discrete Action Spaces. In *2021 IEEE Security and Privacy Workshops (SPW)*, pages 85–91. IEEE. 136, 158, 162, 169

Eykholt, K., Evtimov, I., Fernandes, E., Li, B., Rahmati, A., Xiao, C., Prakash, A., Kohno, T., and Song, D. (2018). Robust physical-world attacks on deep learning visual classification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1625–1634. 100

Fang, Y., Zeng, Y., Li, B., Liu, L., and Zhang, L. (2020). DeepDetectNet vs RLAttackNet: An adversarial method to improve deep learning-based static malware detection model. *PLOS ONE*, 15(4):e0231626. 102, 131, 132, 138, 141, 169

Fang, Z., Wang, J., Geng, J., Zhou, Y., and Kan, X. (2021). A3CMal: Generating adversarial samples to force targeted misclassification by reinforcement learning. *Applied Soft Computing*, 109:107505. 132, 169

Fang, Z., Wang, J., Li, B., Wu, S., Zhou, Y., and Huang, H. (2019). Evading Anti-Malware Engines With Deep Reinforcement Learning. *IEEE Access*, 7:48867–48879. 132, 133, 138, 162, 169

Fellows, M., Mahajan, A., Rudner, T. G., and Whiteson, S. (2019). Virel: A variational inference framework for reinforcement learning. *Advances in neural information processing systems*, 32. 138

Ferrie, P. and Szö, P. (2001). Zmist opportunities. *Virus Bulletin*. 81

Fitzgerald, S., Mathews, G., Morris, C., and Zhulyn, O. (2012). Using NLP techniques for file fragment classification. *Digital Investigation*, 9:S44–S49. The Proceedings of the Twelfth Annual DFRWS Conference. 19

Fleshman, W., Raff, E., Zak, R., McLean, M., and Nicholas, C. (2018). Static malware detection & subterfuge: Quantifying the robustness of machine learning and current anti-virus. In *2018 13th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 1–10. 80

Fryer, D., Strümke, I., and Nguyen, H. (2021). Shapley values for feature selection: The good, the bad, and the axioms. *IEEE Access*, 9:144352–144360. 175

Gibert, D., Fredrikson, M., Mateu, C., Planes, J., and Le, Q. (2022). Enhancing the insertion of nop instructions to obfuscate malware via deep reinforcement learning. *Computers & Security*, 113. arXiv: 2111.09626. 170

Gibert, D., Planes, J., Le, Q., and Zizzo, G. (2023). A wolf in sheep's clothing: Query-free evasion attacks against machine learning-based malware detectors with generative adversarial networks. In *2023 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, pages 415–426. 196, 207, 208

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2020). Generative adversarial networks. *Commun. ACM*, 63(11):139–144. 98

Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014a). Generative adversarial networks. 104, 106

Goodfellow, I. J., Shlens, J., and Szegedy, C. (2014b). Explaining and harnessing adversarial examples. 102, 104, 118

Graham, R., Perriss, R., and Scarsbrook, A. (2005). Dicom demystified: A review of digital file formats and their use in radiological practice. *Clinical Radiology*, 60(11):1133–1140. 14, 42

Harang, R. and Rudd, E. M. (2020). Sorel-20m: A large scale benchmark dataset for malicious pe detection. 132

Hegt, S. (2020). Mark of the web from a red team's perspective. 45

Holland, J. H. (1992). Genetic algorithms. *Scientific American*, 267(1):66–73. 106

Horgan, D., Quan, J., Budden, D., Barth-Maron, G., Hessel, M., van Hasselt, H., and Silver, D. (2018). Distributed prioritized experience replay. *CoRR*, abs/1803.00933. 130

Htet, K. P. (2017). Lazarus group. 46

Jana, S. and Shmatikov, V. (2012a). Abusing file processing in malware detectors for fun and profit. In *2012 IEEE Symposium on Security and Privacy*, pages 80–94. 13, 14, 15, 20, 37, 41

Jana, S. and Shmatikov, V. (2012b). Abusing file processing in malware detectors for fun and profit. In *2012 IEEE Symposium on Security and Privacy*, pages 80–94. 102

Jazi, H. (2021). Lazarus apt conceals malicious code within bmp image to drop its rat. 46

Karresand, M. and Shahmehri, N. (2006). File type identification of data fragments by their binary structure. In *Proceedings of the IEEE Information Assurance Workshop*, pages 140–147. 19

Kiat, N. C., Rosario, A. D., and Co, M. (2021). Seo poisoning and the batloader apt group. 238

Koch, L., Oesch, S., Chaulagain, A., Adkisson, M., Erwin, S., and Weber, B. (2022). Toward the detection of polyglot files. In *Proceedings of the 15th Workshop on Cyber Security Experimentation and Test*, CSET '22, page 120–128, New York, NY, USA. Association for Computing Machinery. 6, 102, 123

Koch, L., Oesch, S., Chaulagain, A., Dixon, J., Dixon, M., Huettal, M., Sadovnik, A., Watson, C., Weber, B., Hartman, J., and Patulski, R. (2024). On the abuse and detection of polyglot files. 6, 210, 217

Kolosnjaji, B., Demontis, A., Biggio, B., Maiorca, D., Giacinto, G., Eckert, C., and Roli, F. (2018a). Adversarial malware binaries: Evading deep learning for malware detection in executables. 36, 68

Kolosnjaji, B., Demontis, A., Biggio, B., Maiorca, D., Giacinto, G., Eckert, C., and Roli, F. (2018b). Adversarial malware binaries: Evading deep learning for malware detection in executables. In *2018 26th European Signal Processing Conference (EUSIPCO)*, pages 533–537. 100, 103, 105

Kozák, M., Jureček, M., Stamp, M., and Troia, F. D. (2024). Creating valid adversarial examples of malware. *Journal of Computer Virology and Hacking Techniques*, pages 1–15. 132, 134, 142, 143, 158, 159, 170, 181, 194, 212

Krcal, M., Svec, O., Balek, M., and Jasek, O. (2018). Deep convolutional malware classifiers can learn from raw executables and labels only. In *International Conference on Learning Representations*. 121

Kreuk, F., Barak, A., Aviv-Reuven, S., Baruch, M., Pinkas, B., and Keshet, J. (2018). Deceiving end-to-end deep learning malware detectors using adversarial examples. *arXiv preprint arXiv:1802.04528*. 80, 104, 187

Kumar, I. E., Venkatasubramanian, S., Scheidegger, C., and Friedler, S. (2020). Problems with shapley-value-based explanations as feature importance measures. In III, H. D. and Singh, A., editors, *Proceedings of the 37th International Conference*

on Machine Learning, volume 119 of *Proceedings of Machine Learning Research*, pages 5491–5500. PMLR. 131, 175, 215

Kurakin, A., Goodfellow, I., Bengio, S., Dong, Y., Liao, F., Liang, M., Pang, T., Zhu, J., Hu, X., Xie, C., et al. (2018). Adversarial attacks and defences competition. In *The NIPS'17 Competition: Building Intelligent Systems*, pages 195–231. Springer. 102

Kusnezov, D., Barsoum, Y., Begoli, E., Henninger, A., and Sadovnik, A. (2023). Risks and mitigation strategies for adversarial artificial intelligence threats: A dhs s&t study. *Department of Homeland Security: Science and Technology*. 88

Labaca-Castro, R., Franz, S., and Rodosek, G. D. (2021). AIMED-RL: Exploring Adversarial Malware Examples with Reinforcement Learning. In Dong, Y., Kourtellis, N., Hammer, B., and Lozano, J. A., editors, *Machine Learning and Knowledge Discovery in Databases. Applied Data Science Track*, Lecture Notes in Computer Science, pages 37–52, Cham. Springer International Publishing. 119, 132, 169

Li, X. and Li, Q. (2021). An IRL-based malware adversarial generation method to evade anti-malware engines. *Computers & Security*, 104:102118. 169

Lucas, K., Sharif, M., Bauer, L., Reiter, M. K., and Shintre, S. (2021). Malware makeover: Breaking ml-based static analysis by modifying executable bytes. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, ASIA CCS '21, page 744–758, New York, NY, USA. Association for Computing Machinery. 122, 127

Maiorca, D., Biggio, B., and Giacinto, G. (2019). Towards adversarial malware detection: Lessons learned from pdf-based attacks. *ACM Computing Survey*, 52. 13

Martignoni, L., Christodorescu, M., and Jha, S. (2007). Omniunpack: Fast, generic, and safe unpacking of malware. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC)*, pages 431–441. IEEE Computer Society. 94

McDaniel, M. and Heydari, M. (2003). Content based file type detection algorithms. In *36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the*. 18, 21

McGraw, G. and Morrisett, G. (2000). Attacking malicious code: A report to the infosec research council. *IEEE Software*, 17(5):33–41. 140

Merabet, H. E. and Hajraoui, A. (2019). Survey of malware detection techniques based on machine learning. *International Journal of Advanced Computer Science and Applications*, 10. 22

Microsoft (2018a). Imagefile object. 46

Microsoft (2018b). Imageprocess object. 46

Mittal, G., Korus, P., and Memon, N. (2020). Fifty: Large-scale file fragment type identification using neural networks. 41, 56

Mittal, G., Korus, P., and Memon, N. (2021). Fifty: Large-scale file fragment type identification using convolutional neural networks. *IEEE Transactions on Information Forensics and Security*, 16:28–41. 19, 21

Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR. 136

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. A. (2013). Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602. 108, 130, 136

Muralidharan, T., Cohen, A., Gerson, N., and Nissim, N. (2022). File packing from the malware perspective: Techniques, analysis approaches, and directions for enhancements. *ACM Computing Surveys*. 95

Nguyen, T. T. and Reddi, V. J. (2023). Deep reinforcement learning for cyber security. *IEEE Transactions on Neural Networks and Learning Systems*, 34(8):3779–3795. 85

of Bits, T. (2022). Polyfile. 56

Or-Meir, O., Nissim, N., Elovici, Y., and Rokach, L. (2019). Dynamic malware analysis in the modern era—a state of the art survey. *ACM Comput. Surv.*, 52(5). 94

Ortiz, M. (2019). HIPAA-protected malware? exploiting dicom flaw to embed malware in ct/mri imagery. *Cylera Labs*. 12, 37, 42

Papernot, N., McDaniel, P., and Goodfellow, I. (2016a). Transferability in machine learning: from phenomena to black-box attacks using adversarial samples. 98, 109, 173

Papernot, N., McDaniel, P., Goodfellow, I., Jha, S., Celik, Z. B., and Swami, A. (2016b). Practical black-box attacks against machine learning. 98

Pappas, V., Polychronakis, M., and Keromytis, A. D. (2012). Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *2012 IEEE Symposium on Security and Privacy*, pages 601–615. 120, 156, 157

Park, S. (2021). Andariel evolves to target South Korea with ransomware. 46

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830. 176

Pierazzi, F., Pendlebury, F., Cortellazzi, J., and Cavallaro, L. (2020). Intriguing properties of adversarial ml attacks in the problem space. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1332–1349. 100

Pontello, M. (2020). Trid file identifier. 56

Popescu, D. (2012a). Hiding malicious content in PDF documents. *CoRR*, abs/1201.0397. 42

Popescu, D.-S. (2012b). Hiding malicious content in pdf documents. *arXiv preprint arXiv:1201.0397*. 13

Popescu, D.-S. (2012c). Hiding malicious content in pdf documents. 37

Poslušnỳ, M. and Kálnai, P. (2020). Rich headers: Leveraging this mysterious artifact of the pe format. In *Proceedings of the 2020 VirusBulletin Conference (VB)*. 91

Quertier, T., Marais, B., Morucci, S., and Fournel, B. (2022). Merlin – malware evasion with reinforcement learning. 130, 134, 170

Quinlan, J. R. (1986). Induction of decision trees. In *Machine Learning: Proceedings of the Tenth International Workshop*, pages 81–106. Morgan Kaufmann Publishers Inc. 174

Raff, E., Barker, J., Sylvester, J., Brandon, R., Catanzaro, B., and Nicholas, C. K. (2018a). Malware detection by eating a whole exe. In *AAAI Workshops*. 22, 23, 52, 173

Raff, E., Barker, J., Sylvester, J., Brandon, R., Catanzaro, B., and Nicholas, C. K. (2018b). Malware detection by eating a whole exe. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*. 103, 109, 121

Raff, E., Fleshman, W., Zak, R., Anderson, H. S., Filar, B., and McLean, M. (2020a). Classifying sequences of extreme length with constant memory applied to malware detection. *arXiv preprint arXiv:2012.09390*. 22, 23, 110, 125

Raff, E. and Nicholas, C. (2020). A survey of machine learning methods and challenges for windows malware classification. *arXiv preprint arXiv:2006.09271*. 22

Raff, E., Zak, R., Munoz, G. L., Fleming, W., Anderson, H. S., Filar, B., Nicholas, C., and Holt, J. (2020b). Automatic Yara Rule Generation Using Biclustering. In *13th ACM Workshop on Artificial Intelligence and Security (AISec'20)*. 62

ReFirmLabs (2021). Binwalk. 38, 56

Ronen, R., Radu, M., Feuerstein, C., Yom-Tov, E., and Ahmadi, M. (2018). Microsoft malware classification challenge. *CoRR*, abs/1802.10135. 114, 115, 121

Rosenberg, I., Shabtai, A., Rokach, L., and Elovici, Y. (2018). Generic black-box end-to-end attack against state of the art api call based malware classifiers. 81, 101, 112, 127, 150, 180

Saxe, J. and Berlin, K. (2015). Deep neural network based malware detection using two dimensional binary program features. *CoRR*, abs/1508.03096. 14

Schulte, E., Brown, M. D., and Folts, V. (2022a). A broad comparative evaluation of x86-64 binary rewriters. In *Proceedings of the 15th Workshop on Cyber Security Experimentation and Test*, pages 129–144. 112

Schulte, E., Brown, M. D., and Folts, V. (2022b). A broad comparative evaluation of x86-64 binary rewriters. In *Proceedings of the 15th Workshop on Cyber Security Experimentation and Test*, pages 129–144. 165

Schulte, E., Dorn, J., Flores-Montoya, A., Ballman, A., and Johnson, T. (2019). Gtirb: intermediate representation for binaries. *arXiv preprint arXiv:1907.02859*. 111

Selvaraju, R. R., Cogswell, M., Das, A., Vedantam, R., Parikh, D., and Batra, D. (2019). Grad-cam: Visual explanations from deep networks via gradient-based localization. *International Journal of Computer Vision*, 128(2):336–359. 117

Sewak, M. (2019). *Deep Q Network (DQN), Double DQN, and Dueling DQN: A Step Towards General Artificial Intelligence*, pages 95–108. 131

Sewak, M., Sahay, S. K., and Rathore, H. (2022). Deep reinforcement learning for cybersecurity threat detection and protection: A review. In Krishnan, R., Rao, H. R., Sahay, S. K., Samtani, S., and Zhao, Z., editors, *Secure Knowledge Management In The Artificial Intelligence Era*, pages 51–72. Springer International Publishing. 85

Shah, S. (2015). Weaponized polyglots as browser exploits. *International Journal of Proof-of-Concept or Get The Fuck Out*. 12, 15

Shapley, L. S. (1953). A value for n-person games. In Kuhn, H. W. and Tucker, A. W., editors, *Contributions to the Theory of Games (Vol. 2)*, pages 307–317. Princeton University Press, Princeton, NJ. 131, 174

Sharif, M., Lanzi, A., Giffin, J., and Lee, W. (2009). Automatic reverse engineering of malware emulators. In *2009 30th IEEE Symposium on Security and Privacy*, pages 94–109. 166

Sharif, M., Lucas, K., Bauer, L., Reiter, M., and Shintre, S. (2019). Optimization-guided binary diversification to mislead neural networks for malware detection. 120, 122, 127, 157, 217, 219

Sheehan, C. (2007). Pimp my pe: Parsing malicious and malformed executables. 159

Song, W., Li, X., Afroz, S., Garg, D., Kuznetsov, D., and Yin, H. (2020). Automatic generation of adversarial examples for interpreting malware classifiers. *ArXiv*, abs/2003.03100. 119, 122, 123, 134, 142, 143, 194, 212

Song, W., Li, X., Afroz, S., Garg, D., Kuznetsov, D., and Yin, H. (2021). MAB-Malware: A Reinforcement Learning Framework for Attacking Static Malware Classifiers. *arXiv:2003.03100*. arXiv: 2003.03100. 101, 119, 124, 130, 132, 134, 142, 146, 169, 188

Souri, A. and Hosseini, R. (2018). A state-of-the-art survey of malware detection approaches using data mining techniques. *Human-centric ComputComputing and Information Sciences*, 8. 22

Suciu, O., Coull, S. E., and Johns, J. (2019). Exploring adversarial examples in malware detection. In *2019 IEEE Security and Privacy Workshops (SPW)*, pages 8–14. IEEE. 104, 118, 126

Sundararajan, M., Taly, A., and Yan, Q. (2017). Axiomatic attribution for deep networks. 115

Sundareswaran, S. and Squicciarini, A. C. (2010). Image repurposing for gifar-based attacks. 14, 20

Sutton, R. S., Bach, F., and Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT Press Ltd. 107, 108, 129, 130, 132, 135, 136

Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., and Fergus, R. (2013). Intriguing properties of neural networks. 2, 88, 98, 102, 211

Tan, H., Luo, Q., Li, J., and Zhang, Y. (2024). LLM4Decompile: Decompiling binary code with large language models. *arXiv preprint arXiv:2403.05286*. 216

Team, C. P. R. (2022a). October's most wanted malware: Agenttesla knocks formbook off top spot and new text4shell vulnerability disclosed. 45

Team, C. R. (2022b). Can you trust a file's digital signature? new zloader campaign exploits microsoft's signature verification putting users at risk. 238

Technologies, O. (2024). Code virtualizer overview. 166

Thompson, W. R. (1933). On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4):285–294. 131

van Hasselt, H., Guez, A., and Silver, D. (2015). Deep reinforcement learning with double q-learning. 130, 136

Veenman, C. J. (2007). Statistical disk cluster classification for file carving. In *Third International Symposium on Information Assurance and Security*, pages 393–398. 19

Vuksan, M. and Pericin, T. (2011). Constant insecurity: Things you didn't know about (pe) portable executable file format. 91, 143

Wang, Z., Bapst, V., Heess, N., Mnih, V., Munos, R., Kavukcuoglu, K., and Freitas, N. (2016a). Sample efficient actor-critic with experience replay. 130, 136

Wang, Z., Schaul, T., Hessel, M., van Hasselt, H., Lanctot, M., and de Freitas, N. (2016b). Dueling network architectures for deep reinforcement learning. 131

Wenzl, M., Merzdovnik, G., Ullrich, J., and Weippl, E. (2019). From hack to elaborate technique—a survey on binary rewriting. *ACM Computing Surveys (CSUR)*, 52(3):1–37. 112

Williams, R. J. (1992). Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256. 130

Wu, C., Shi, J., Yang, Y., and Li, W. (2018). Enhancing Machine Learning Based Malware Detection Model by Reinforcement Learning. In *Proceedings of the 8th International Conference on Communication and Network Security - ICCNS 2018*, pages 74–78, Qingdao, China. ACM Press. 102, 130, 131, 133, 136, 137, 138, 158, 162, 169

Zhan, D., Bai, W., Liu, X., Hu, Y., Zhang, L., Guo, S., and Pan, Z. (2023). Pspmal: Evading malware detection via prioritized experience-based reinforcement learning with shapley prior. In *Proceedings of the 39th Annual Computer Security Applications Conference*, pages 580–593. 131, 132, 158, 162, 165, 170, 175

Zhang, L., Liu, P., Choi, Y., and Chen, P. (2022). Semantics-preserving reinforcement learning attack against graph neural networks for malware detection. *IEEE Transactions on Dependable and Secure Computing*. 85, 110, 111, 164, 165, 170

Zhong, F., Cheng, X., Yu, D., Gong, B., Song, S., and Yu, J. (2020). Malfox: Camouflaged adversarial malware example generation based on c-gans against black-box detectors. *CoRR*, abs/2011.01509. 129

Zhong, F., Hu, P., Zhang, G., Li, H., and Cheng, X. (2022). Reinforcement learning based adversarial malware example generation against black-box detectors. *Computers & Security*, 121:102869. 128, 129, 137, 150, 162, 166, 170

Zhu, S., Shi, J., Yang, L., Qin, B., Zhang, Z., Song, L., and Wang, G. (2020). Measuring and modeling the label dynamics of online Anti-Malware engines. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2361–2378. USENIX Association. 164

Appendix

Batloader/Zloader Cyber Attack Chain

Batloader and Zloader are two very similar pieces of malware that are used to gain initial access Kiat et al. (2021); Team (2022b). The full attack chain is presented in Figure 6.1; however, our discussion will focus on the role of the polyglot within that chain. This polyglot is formed by combining an HTA file with a Windows PE file.

Windows PE files are the default executable for the Windows ecosystem. Since their format specification requires the bytes "MZ" to be present at offset zero, this format must be the first—by offset—ingredient in a polyglot in order to preserve functionality. PE polyglots can be created via the *cave* or *stack* method. The *cave* method places the second file in a slack region of the PE. Candidate locations include the DOS Stub, after the last section table entry, or in the padding space after each section assuming the chosen region is large enough to contain the second file. The *stack* method simply appends the second file to the end (also referred to as the overlay) of the PE file.

In this particular example, an HTA file is added to the signature section of the PE file. Rather ironically, CVE-2020-1599 allows malware authors to add contents to the signature section without invalidating the signature since the contents of this area need to be writable in order to store the calculated signature.

By preserving the validity of the rest of the file, the infected PE is able to operate with a higher degree of trust than it would otherwise Díaz (2022). Note that, although Microsoft addressed this vulnerability by creating an option to disallow extraneous

data in the signature section, this option is turned off by default. The higher degree of trust accorded to the signed PE allows the covert HTA to execute the final payloads in the Batloader/Zloader attack chain.

How File Formats Enable Polyglot Capabilities

The following sections detail how each overt format was used in combination with a covert format to surreptitiously execute or stage a malicious payload. This is followed by details on the roles filled by polyglot files in notable cyber-attack chains.

HTA

HTML Application (HTA) support in Windows is intended to make the Internet Explorer browser a Windows desktop development platform. It gives developers the flexibility to create full-scale applications using web-based technologies, such as HTML, JavaScript, and Visual Basic Script (VBScript) without following the strict security model of the browser.

HTA contents are not executed directly; rather, they are fed to MSHTA.exe, a trusted signed Microsoft binary packaged with the Windows ecosystem, which then executes the contents. A few features of HTAs have attracted threat actors to include HTA files within their cyber-attacks.

1. HTA files are loaded by a trusted, Microsoft-signed application, allowing attackers to bypass restrictions on application execution.

2. HTA files can be loaded remotely, allowing malicious activities to be run without even being copied to the target's disk.

3. MSHTA.exe has a generous parser, and does not require the HTA file signature *<hta:application* for execution. MSHTA.exe will attempt to execute any HTML or VBScript/JavaScript code passed to the binary. This can make reliable identification of HTA files or fragments challenging for signature-based tools.

The extensive usage of HTA files by attackers has led to an arms race between new detection methods and evasion techniques, with polyglots being one of the latest developments. A simple HTA attack can be detected using a rule which checks whether a MSHTA.exe process has been launched with an HTA file as input. This detection can be evaded by renaming/moving MSHTA.exe to a new name/location and by turning the HTA file into a polyglot that masquerades as a different file format. Since MSHTA.exe skips all data it does not understand, HTA files can be combined with a wide variety of file formats for obfuscation.

PHP and PHAR

PHP is a popular programming language for web applications that provides dynamic rendering of web pages, database access, and many other features. The PHAR file format is the archive format of the language, comparable to JAR files within the Java ecosystem. As with HTA files, PHP has a generous parser that ignores a wide variety of syntax errors and invalid characters. Invalid characters are ignored until valid PHP code is found. Therefore, PHP and PHAR files can readily be combined with a number of file formats.

Polyglots whose covert format is PHP or PHAR typically utilize an image format (JPEG, PNG, GIF, BMP) as their overt component, likely due to the prevalence and (possibly) lower level of scrutiny applied to images within web application file structures. Since image files are commonly publicly accessible through file upload services, PHP and PHAR polyglots can serve as covert methods for staging and then executing malicious code on web servers. A web server's logs could merely show that a customer accessed a stored image when in reality they remotely executed malicious activity. Additionally, web servers that attempt to block malicious activity by preventing the upload of certain file formats are vulnerable to polyglots that masquerade as an approved format.

JAR

The Java community created the Java Archive (JAR) to package a Java application, Java libraries, and other application resources in a single file. JAR files are an extension of the common Zip format. The contents of a Zip file are located by first scanning the end of the file for the *central directory* which contains the relative offsets to the compressed files held within the archive, allowing another file to be prepended to an archive file without invalidating the data already contained in the archive.

Recently, threat actors created polyglots using JAR files. One possible reason is the discovery of the Windows vulnerability, CVE-2020-1464. This was a weakness in Windows Installer (MSI) files and is related to the manner by which their digital signature is validated within the Windows operating system.

Normally, an MSI file is cryptographically signed by the developer, allowing an end-user to verify that the MSI not only came from the expected developer, but also has not been altered in transit. However CVE-2020-1464 allowed an attacker to append a malicious JAR file to the end of an MSI file without invalidating the signature of the MSI file, creating a polyglot with a covert format of JAR and an overt format of MSI. This vulnerability remained unpatched in the Windows operating system for at least two years.

Zip and RAR

The survey did not produce many instances of Zip and RAR polyglots. This may be due to the deletion of Zip and RAR polyglots once their contents have been extracted. In the attack chains observed with archive format polyglots, polyglot files with a covert format of Zip or RAR allowed covert transfer of the polyglot archive's malicious contents thanks, typically, to their image-based overt format.

Note, RAR files are not derived from Zip files; they are a distinct archive format. That said, RAR and Zip files both tolerate prepended data. Whereas Zip files are

read from the bottom up, RAR files are read forward, skipping extraneous content until the RAR header is found.

JavaScript

The JavaScript language is a ubiquitous web technology used to build many web applications and is supported in all modern browsers. This provides a large attack surface for attackers. The survey discovered at least one instance of an attacker using a polyglot with a JavaScript covert format and an image-based overt format to infiltrate advertisement networks.

Normally, reputable advertising companies restrict scripts in their advertisements to avoid sending end-users malicious code. However, this polyglot could bypass script detection without loss of functionality by posing as an advertising image. We were unable to get the sample for this attack.

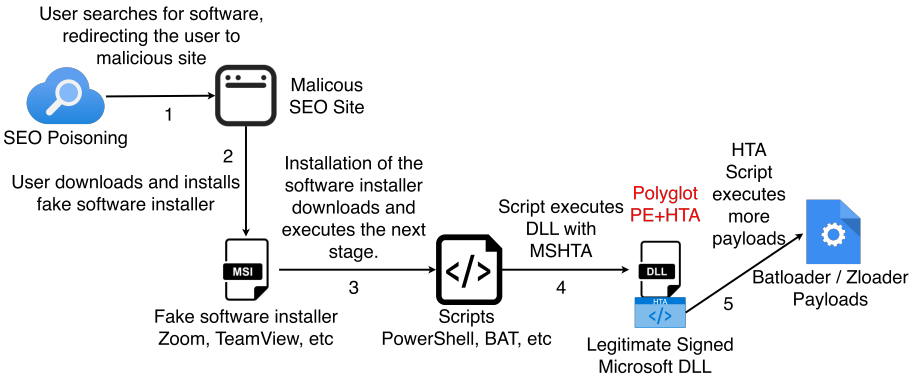


Figure 6.1: Batloader/Zloader Attack Chain

Table 6.1: Malicious Polyglot Hashes

Malware Name	Formats	File Name	File Hash (SHA-256)
IcedId	CHM+HTA	pss10r.chm	3d279aa8f56e468a014a9163625409759 58b9e9172d658eb57065a8a230632fa
Batloader	PE+HTA	AppResolver.dll	1258fb78dd50f6c12c3181cc5c1362dc9d70ca46c5fd7e6af4880ee6d6d9e7a2
Batloader	PE+HTA	AppResolver.dll	588af958bc4365ecff4264a9fb75351eaee1ca9d0672c3040a77f979795219bd
Batloader	PE+HTA	AppVEntStreamingManager.dll	3ec8b76ac735348db87bd0bf766554a2cb280f94d12dad8a159e917e00ab28f2
ZLoader	PE+HTA	AppVSentinel.dll	64a0a6ac17128ce7fd4dc34556bfe4736900121e5766557bceeac0cce99fbe21
ZLoader	PE+HTA	AppResolver.dll	89ccde97787a3eb0f9de38ab51c9f3278a3b18531b0fa468f08b55a133263b1c
ZLoader	PE+HTA	AppResolver.dll	950ad539dfc8e16c07d24dbb37ae19daa0b2f32164ba0cb3c81fa7e689f274e1
ZLoader	PE+HTA	AppResolver.dll	a187c9bb2a8bc29184bd18d6f515532d0f9b3f97b53f0ec6347b9982c4dff00f
ZLoader	PE+HTA	AppResolver.dll	c1a34057b31dd53e227a7001a7f0860e553b7efdb9ea2e9ec3b80221266b7d51
ZLoader	PE+HTA	advapi32.dll	d1a1381c1f02abaa3449451136c1d1054ed728183482971 13c135e8211173b3f
ZLoader	PE+HTA	AppResolver.dll	eb7354a95762565558d46753caf0c0d4dd09e1f358d564ae034b64446599e907
Lazarus	BMP+HTA	imgFBE0.tmp	fe16b1dc30ee50ab126129c7fc0f2e6932083d4429241707d8046760c6b25042
Lazarus	BMP+HTA	image003.zip - undetected	c9803b32365f4870d4ca833eb418eb845f16c4ec1628253a152667d935d9985b
Lazarus	BMP+HTA	image003.zip - undetected	a95a3fd25ab87c5010d42fe0131338b78187672dd6dc213af4253ef5db494591
Lazarus	BMP+HTA	image003.zip	888cfc87b44024c48eed794cc9d6dea9f6ae0cc3468dee940495e839a12ee0db
PHP Shell	JPEG+PHP	63f4c7b002cc47.jpg	4e26b08cce3fbd04fb9d954e1fa6a72d91f909015e7564aae9570aee26e8efd6
PHP Shell	JPEG+PHP	simp.php.gif	47102e200c35185654e74237a838e4c6b484cadd5a97d77aa7ad633b4f83ba62
PHP Shell	JPEG+PHP	images.jpg	0b5fd1d621affa41ebe811a39c085d62be489c55e26705b1db61accaa1dbcb6a
PHP Shell	JPEG+PHP	001.swf.jpeg	71f463e8d5c0f7ec6221a1cb9d5683766d5f7270ca80395bee5d0d00ec4ba0f3
PHP Shell	JPEG+PHP	20190225150235_34013.php	5f8e797b0f2b2efee4839841cc7b597f80b8b6f1558ec18b43a834e4bd540fdb
PHP Shell	JPEG+PHP	v1QR1M.gif	e028dc0e26b03a8a9cd5de11515f485dbaa57b721cb4ff4b1ffa115e64459eb9
PHP Shell	GIF+PHP	Adipati.php	b660e691007a1fd8301f39782019a5f7bee6fd7dea18545e372a67014cee4c42
PHP Shell	JPEG+PHP	Logo_Coveright.jpg	ab85eb33605f3013989f4e8a9bfd5e89dd82d1f80231d4e4a2ceb82744bf287c
PHP Shell	JPEG+PHP	ce167d905d117823d780e188002b3120.jpg	39588ed13465b15ec59ec35a885de028d0b6537cf6410c96402adfe1053694d6
PHP Shell	PNG+PHP	in1.png	57507a3db555182882c0c335b0b943ee2f977a1a9cf973be070fa9db6491cdf5
SyncCrypt	JPEG+ZIP	003_JPG.arrival.jpg	c6565d22146045e52110fd0a13eba3b6b63fbf6583c444d7a5b4e3a368cc4b0d
DarkTrack RAT	PNG+RAR	darknet.jpg	ee0c0be30ba2875a2bc7813ae80814659ce35988fbd9d5232950ed7722b89a9a
JAR/MSI	MSI+JAR	488adc.msi	dd71284ac6be9758a5046740168164ae76f743579e24929e0a840afd6f2d0d8e
Ratty	MSI+JAR	29-05-2020.jar	90f613caa131c663e32aabc31b5fccc99edcfa874110d51cd627531d3a67b16d
Ratty	MSI+JAR	6afad7.msi	04a3cad80470a085b6ef57a7e1007049a29863a94fe76f93be1f2a0c54da99d6

Table 6.2: Sources for Malicious Polyglot Usage in the Wild

Title	Publisher	Date Published	URL
New Banking Trojan IcedID Discovered by IBM X-Force Research	Security Intelligence	13 November 2017	https://securityintelligence.com/new-banking-trojan-icedid-discovered-by-ibm-x-force-research/
More Than Meets the Eye: Exposing a Polyglot File That Delivers IcedID	Palo Alto Networks	27 September 2022	https://unit42.paloaltonetworks.com/polyglot-file-icedid-payload/
Zoom For You — SEO Poisoning to Distribute BATLOADER and Atera Agent	Mandiant	1 February 2022	https://www.mandiant.com/resources/blog/seo-poisoning-batloader-atera
Can You Trust a File's Digital Signature? New Zloader Campaign exploits Microsoft's Signature Verification putting users at risk	Check Point Research	5 January 2022	https://research.checkpoint.com/2022/can-you-trust-a-files-digital-signature-new-zloader-campaign-exploits-microsofts-signature-verification-putting-users-at-risk/
BATLOADER: The Evasive Downloader Malware	VMWare	14 November 2022	https://blogs.vmware.com/security/2022/11/batloader-the-evasive-downloader-malware.html
Monitoring malware abusing CVE-2020-1599	VirusTotal	7 January 2022	https://blog.virustotal.com/2022/01/monitoring-malware-abusing-cve-2020-1599.html
Lazarus APT conceals malicious code within BMP image to drop its RAT	MalwareBytes	19 April 2021	https://www.malwarebytes.com/blog/threat-intelligence/2021/04/lazarus-apt-conceals-malicious-code-within-bmp-file-to-drop-its-rat
Andariel evolves to target South Korea with ransomware	Kaspersky	15 June 2021	https://securelist.com/andariel-evolves-to-target-south-korea-with-ransomware/102811/
LNK HTA Polyglot	Hatching	12 November 2018	https://hatching.io/blog/lnk-hta-polyglot/
PHP WebShell Malware using Image Files	ASEC	9 December 2020	https://asec.ahnlab.com/en/18861/
Hiding Webshell Backdoor Code in Image Files	Trustwave	11 October 2013	https://www.trustwave.com/en-us/resources/blogs/spiderlabs-blog/hiding-webshell-backdoor-code-in-image-files/
Malware in Images: When You Can't See "the Whole Picture"	Reversing Labs	2 March 2021	https://blog.reversinglabs.com/blog/malware-in-images
Picture perfect: How JPG EXIF data hides malware	Cisco	24 July 2019	https://umbrella.cisco.com/blog/picture-perfect-how-jpg-exif-data-hides-malware
Lab: Remote code execution via polyglot web shell upload	PortSwigger	Unknown	https://portswigger.net/web-security/file-upload/lab-file-upload-remote-code-execution-via-polyglot-web-shell-upload
Playing with GZIP: RCE in GLPI (CVE-2020-11060)	Almond	14 May 2020	https://offsec.almond.consulting/playing-with-gzip-rce-in-glpi.html
It's a PHP Unserialization Vulnerability Jim, but Not as We Know It	Blackhat	9 August 2018	https://i.blackhat.com/us-18/Thu-August-9/us-18-Thomas-Its-A-PHP-Unserialization-Vulnerability-Jim-But-Not-As-We-Know-It.pdf
CVE-2022-41343 - RCE via Phar Deserialization	Tanto	6 October 2022	https://tantosec.com/blog/cve-2022-41343/
Taiwan Heist: Lazarus Tools and Ransomware	BAE Systems	16 October 2017	https://baesystemsai.blogspot.com/2017/10/taiwan-heist-lazarus-tools.html
SyncCrypt Ransomware Hides Inside JPG Files, Appends .KK Extension	Bleeping Computer	16 August 2017	https://www.bleepingcomputer.com/news/security/synccrypt-ransomware-hides-inside-jpg-files-appends-kk-extension/
DarkTrack RAT – New Variant Thumbing a Ride in PNG Files	SECTRIO	25 August 2020	https://www.subexsecure.com/pdf/malware-reports/August-2020/DarkTrack-Report.pdf

Table 6.3: Sources for Malicious Polyglot Usage in the Wild Continued

Title	Publisher	Date Published	URL
Distribution of malicious JAR appended to MSI files signed by third parties	VirusTotal	15 January 2019	https://blog.virustotal.com/2019/01/distribution-of-malicious-jar-appended.html
Interesting tactic by Ratty & Adwind for distribution of JAR appended to signed MSI – CVE-2020-1464	Security-in-bits	28 June 2020	https://www.securityinbits.com/malware-analysis/interesting-tactic-by-ratty-adwind-distribution-of-jar-appended-to-signed-msi/
Microsoft Put Off Fixing Zero Day for 2 Years	Krebs on Security	17 August 2020	https://krebsonsecurity.com/2020/08/microsoft-put-off-fixing-zero-day-for-2-years/
GlueBall: The story of CVE-2020–1464	Tal Be'ery	16 August 2020	https://medium.com/@TalBeerySec/glueball-the-story-of-cve-2020-1464-50091a1f98bd
Uncovering and Disclosing a Signature Spoofing Vulnerability in Windows Installer: CVE-2021-26413	Okta	19 April 2021	https://sec.okta.com/articles/2021/04/uncovering-and-disclosing-signature-spoofing-vulnerability-windows
Hacking Group Using Polyglot Images to Hide Malvertising Attacks	Devcon	24 February 2019	https://www.devcondetect.com/blog/2019/2/24/hacking-group-using-polyglot-images-to-hide-malvertsing-attacks
Bypassing Content Security Policy with a JS/GIF Polyglot	Ajin Abraham	10 June 2015	https://ajinabraham.com/blog/bypassing-content-security-policy-with-a-jsgif-polyglot
WordPress Postie 1.9.40 Plugin - Persistent Cross-Site Scripting Exploit	Vulners	16 January 2020	https://vulners.com/zdt/1337DAY-ID-33819
CVE-2021-27190 – PEEL SHOPPING	Secuneus	11 February 2021	https://www.secuneus.com/cve-2021-27190-peel-shopping-ecommerce-shopping-cart-stored-cross-site-scripting-vulnerability-in-address/

Vita

Luke is a dude who does things. He's good at finding things so dumb no one tried them before and using those things to break fancy things made by much smarter people. He started out as a construction worker whose law enforcement dad taught him to find the vulnerabilities in social/physical spaces. He went back to school and found that a solid grounding in mathematics and low-level programming allowed him to find the vulnerabilities in cyber systems that are increasingly a required component of modern society.

Education:

- 2009: A.S., General Studies, Pellissippi State Community College, 4.0 GPA
- 2020: B.A., Mathematics, Maryville College, 4.0 GPA
- 2024: Ph.D., Data Science and Engineering, University of Tennessee, 4.0 GPA

Experience:

- 2010-2018: Unskilled labor with various construction, industrial maintenance, and landscaping companies (i.e., the rock-shoveling, scaffold-building, rough-carpentry years)
- 2019: SULI with Cybersecurity Research Group at ORNL
- 2020-2023: GRA at ORNL via Bredesen Center
- 2024-Present: Staff Scientist with Emerging Cyber Systems Research Group at ORNL

Field of Study: The intersection of data science and cybersecurity, with a specific focus on red-teaming AI systems designed to detect malicious activity. On the cybersecurity side, this work requires intimate knowledge of file-format specifications and existing malware obfuscation methods paired with brute-force experimentation to develop/understand functionality-preserving perturbations in malware. On the data science side, this work requires an understanding of feature spaces, model architectures, explainability metrics, and adversarial methods.