# DEEP LEARNING FOR OBFUSCATED CODE ANALYSIS

Alexander Shroyer

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Department of Intelligent Systems Engineering,
Indiana University
December 2023

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

Doctoral Committee

D. Martin Swany, Ph.D.

Ariful Azad, Ph.D.

XiaoFeng Wang, Ph.D.

David Crandall, Ph.D.

Date of Defense: December $4^{th}$, 2023

For my family, who encouraged me to begin this journey, supported me through the ups and downs, and are possibly even more excited than I am about its conclusion.

## ACKNOWLEDGMENTS

First and foremost, I am grateful to my doctoral committee for deftly balancing their roles as mentors while giving me the freedom to explore this topic and learn the art and craft of research for myself. You helped me ask the right questions and saved me from getting stuck multiple times. I also want to thank my advisor Martin Swany for not only taking a chance on me as a graduate student, and not only for encouraging me to "go for the Ph.D!" when I initially expressed interest in graduate school, but also being a steadfast guide and friend over these years.

Several of my colleagues and peers also deserve special acknowledgment: my collaborators Vafa Andalibi and Paventhan Vivekanandan have been a consistent joy to work with as we brainstormed and discussed solutions to problems large and small within our small corner of the broader field of embedded system security. At critical moments during my studies I was fortunate to receive sage wisdom from Thomas Sterling, Luke D'Alessandro, Ryan Newton, and Jeremy Siek. Dynamically shifting between roles of friends, colleagues, students, and even teachers, I am thankful for the camaraderie of Jeremy Musser, Prateek Srivastava, Veda Koraganji. Jean-Paul van Oosten also shared some great suggestions that helped me with experimental model implementations in Chapter 4. Andreas Bueckle helped me improve the confusion matrix plots.

Finally, I would never have started this journey without Bronwyn. Thank you for your encouragement, support, perspective, and fastidious observance of proper comma placement.

Alexander Shroyer

DEEP LEARNING FOR OBFUSCATED CODE ANALYSIS

Modern software development relies increasingly on third-party code dependencies, which enables rapid development but also increases risk of introducing bugs, malware, or unauthorized intellectual property. The goal of this dissertation is to reduce these risks making them easier to detect. Determining the meaning of an arbitrary program reduces to solving the halting problem, which is provably impossible. Instead, this work focuses on a narrower scope: to assign a similarity metric between a known program and an unknown one.

To be able to quantify the distance between two programs, one must take into account slight variations in programs due to diverse compilation, whether debug symbols are stripped, or even intentional obfuscation. We address this variation by adding diversity to our training sample data through diverse compilation and deliberate obfuscation. These methods preserve the syntactic and structural qualities of valid code and permit augmentation of sparse datasets on a large scale.

We train a variety of models to classify programs in the augmented training data. These trained models can now predict which parts of unknown programs are most similar to the training programs. In this work we train on standard library functions originally implemented in the C programming language within the musl library. This forms the basis of a novel method which can be applied to other codebases in order to quickly scan for similar examples in unfamiliar code.

## TABLE OF CONTENTS

**Chapter 1. Introduction** . . . . . 1

1.1 Security is easy to sacrifice . . . . . 1

1.2 Costs and benefits of external dependencies . . . . . 2

1.3 Make it easier to do the right thing . . . . . 2

1.4 Choose one part of the larger problem . . . . . 3

1.5 Binary analysis . . . . . 3

1.6 Obfuscation as data augmentation . . . . . 4

1.7 Contributions of this dissertation . . . . . 5

**Chapter 2. Data Augmentation** . . . . . 6

2.1 Introduction . . . . . 6

2.2 Related Work . . . . . 8

2.3 Methods . . . . . 10

2.3.1 Data Augmentation by Source Code Obfuscation . . . . . 11

2.3.2 Semantics-Preserving Transformations . . . . . 13

2.3.3 Test Programs . . . . . 13

2.3.4 Data Augmentation Types . . . . . 14

2.3.5 Machine Learning Model . . . . . 16

2.4 Results . . . . . 21

2.5 Conclusions and Future Work . . . . . 24

**Chapter 3. Detecting Standard Library Functions in Obfuscated Code . . . . . 27**

3.1 Introduction . . . . . 27

3.2 Related Work . . . . . 29

3.3 Methods . . . . . 31

3.3.1 Data Augmentation Through Obfuscation . . . . . 32

3.3.2 Data Preprocessing . . . . . 34

3.3.3 PV-DM Voting Classifier . . . . . 36

3.3.4 Graph Metadata Classifier . . . . . 36

3.4 Discussion . . . . . 39

3.5 Results . . . . . 40

3.6 Conclusions and Future Work . . . . . 44

**Chapter 4. Function Classification for Obfuscated Binary Code . . . . . 47**

4.1 Introduction . . . . . 47

4.1.1 Related work . . . . . 49

4.1.2 Research questions . . . . . 51

4.2 Methods . . . . . 52

4.2.1 Dataset . . . . . 52

4.2.2 Models . . . . . 60

4.3 Results and Discussion . . . . . 63

4.3.1 Block-Walk Embedding . . . . . 65

4.3.2 Function Detection . . . . . 70

4.4 Conclusions . . . . . 74

**Chapter 5. Related Work** . . . . . 76

5.1 Introduction . . . . . 76

5.1.1 Threats Affecting Software and Hardware . . . . . 76

5.1.2 Vulnerabilities Are Bugs . . . . . 77

5.1.3 Software Compilation . . . . . 79

5.2 Motivation . . . . . 79

5.2.1 Ransomware . . . . . 79

5.2.2 Cybersecurity . . . . . 81

5.2.3 IoT Devices . . . . . 82

5.2.4 Physical Security . . . . . 83

5.2.5 Supply Chain . . . . . 83

5.2.6 Software "Toolchains" . . . . . 84

5.3 Challenges . . . . . 85

5.4 Opportunities . . . . . 86

5.5 Software Vulnerabilities and Mitigations . . . . . 87

5.5.1 Deobfuscation . . . . . 88

5.5.2 Concolic Testing . . . . . 89

5.5.3 Bug Signatures . . . . . 90

5.5.4 Binary Analysis Techniques . . . . . 90

5.6 Software Compilation . . . . . 92

5.6.1 Compiler Construction . . . . . 92

5.6.2 Nanopass Compiler Framework . . . . . 94
5.6.3 LLVM and Intermediate Languages . . . . . 94
5.6.4 Intermediate Representations for Different Systems . . . . . 95
5.6.5 OpenCL . . . . . 95
5.6.6 Compiler Vulnerabilities . . . . . 96
5.7 Firmware . . . . . 98
5.8 Control Flow Graph (CFG) Analysis . . . . . 99
5.8.1 CFG Algorithm Comparison . . . . . 99
5.8.2 discovRE . . . . . 100
5.8.3 Zynamics Bindiff . . . . . 100
5.9 Machine Learning for Software Reverse Engineering . . . . . 101
5.9.1 CNN and Data Augmentation . . . . . 101
5.9.2 Siamese Networks for Similarity . . . . . 101
5.9.3 Asm2Vec . . . . . 102
5.9.4 Malware Detection from Small Samples . . . . . 103
5.9.5 Graph Embedding . . . . . 103
5.9.6 Language-Independent Algorithm Detection . . . . . 105
5.9.7 Deep Networks . . . . . 105
5.9.8 Attention for Graph Neural Networks . . . . . 105
5.9.9 Language-Independent Code Semantic Learning . . . . . 105
5.10 Symbolic Execution . . . . . 106
5.11 Graph Algorithms . . . . . 106
5.11.1 Graph Isomorphism . . . . . 107

5.12 Human Factors . . . 108
5.13 Simulation vs. Formal Verification [127] . . . 108
5.13.1 Model-Based Formal Verification . . . 109
5.13.2 Proof-Theoretic Methods . . . 109
5.13.3 Symbolic State Traversal . . . 109
5.13.4 Symbolic Simulation . . . 110
5.13.5 Summary of [127] . . . 110
5.14 OpenCL [106] . . . 111
5.14.1 How OpenCL Relates to Hardware/Software EDA Security . . . 111
5.15 High Level Synthesis . . . 111
5.16 Alternatives to HLS . . . 112
5.17 Graph Representation in EDA Tools . . . 113
5.17.1 In FPGA Synthesis . . . 113
5.17.2 In Compilers . . . 114
5.17.3 In Both . . . 115

**Chapter 6. Discussion** . . . 116
6.1 Results and interpretation . . . 116
6.2 Limitations . . . 118
6.3 Unexpected results . . . 121
6.4 Future research . . . 122

**Chapter 7. Conclusions** . . . 123
7.1 Structure . . . 123

7.2 Representation . . . . . 123

7.3 Application . . . . . 124

**References** . . . . . 125

**Curriculum Vitae**

## LIST OF TABLES


2.1 selected `musl` library functions . . . . . 10

2.2 Assembly augmentation by transposing lines . . . . . 19

3.1 Comparison of Current Approaches . . . . . 30

3.2 Functions of interest (compatible with Tigress) . . . . . 31

3.3 Individual model accuracy on test set . . . . . 40

3.4 Classifier results: new data . . . . . 41

3.5 Comparison with Asm2Vec . . . . . 43

4.1 Train and test combinations . . . . . 52

4.2 Addresses and partial listing of instructions for disassembled object file. Block number and addresses **bold** for the random walk (1,2,4,6,8). The last column indicates whether the random walk takes the conditional jump. . . . . 56

4.3 Total numbers of files and their basic blocks for each function. Subscripts $o$ and $p$ indicate **Obfuscated** and **Plain** data, respectively. . . . . 57

4.4 Macro-averaged F1 scores (train Obfuscated / test Obfuscated) . . . . . 66

4.5 Macro-averaged F1 scores (train Plain / test Obfuscated) . . . . . 66

4.6 Macro-averaged F1 scores (train Subset / test Obfuscated) . . . . . 67

4.7 Macro-averaged F1 scores (train Obfuscated / test Plain) . . . . . 67

4.8 Macro-averaged F1 scores (train Plain / test Plain) . . . . . 68

4.9 Macro-averaged F1 scores (train Subset / test Plain) . . . . . 68

4.10 F1 scores per-function after 150 training epochs . . . . . 71

## LIST OF FIGURES


2.1 Generating diverse binaries from single sources . . . . . 12

2.2 `pow.c` source code . . . . . 13

2.3 Training loss for data augmentation . . . . . 17

2.4 NLP-style augmentation maintains function size distribution . . . . . 20

2.5 t-SNE of *Plain* and *NLP*-augmented data . . . . . 22

2.6 t-SNE, labeled . . . . . 23

2.7 t-SNE of NLP augmented data . . . . . 23

2.8 Mean function similarity . . . . . 24

3.1 C source code for `abs` function . . . . . 32

3.2 Selected `abs` assembly code variations . . . . . 32

3.3 Graph representation of `Split` . . . . . 35

3.4 Normalized assembly tokens . . . . . 35

3.5 First 2 principal components of **Word2Vec** embedding (with labels) . . . . . 37

3.6 **Graph reduction**. **(a)** shows the original graph; **(b)** condenses strongly connected components and self-loops as per [52]; **(c)** condenses as in (b) and also prunes edges which are not part of the minimum spanning tree; **(d)** applies (c) twice. Each view preserves the relative positioning from the original graph, scaled to fit the new graph. . . . . 38

3.7 Confusion matrix for graph metadata classifier (validation data) . . . . . 42

3.8 Confusion matrix for graph metadata classifier (test data) . . . . . 43

3.9 Mistaken function example . . . . . 44

3.10 Jump-Label distribution . . . . . 46

4.1 Dataset generation stages . . . . . 53
4.2 Distribution of object file sizes . . . . . 58
4.3 Byte counts . . . . . 59
4.4 Block-Walk Embedding model; `BATCH` dimension depends on the batch size for each dataset . . . . . 64
4.5 Confusion matrices for different models and train/test configurations . . . . . 69
4.6 Training the Block-Walk Embedding model . . . . . 70
4.7 Function predictions in "petya0" binary . . . . . 73
4.8 Function predictions in assorted programs ($T = 0.7, W = 30, S = 20$) . . . . . 73

5.1 New Malware Identification 2011-2020 [7] . . . . . 80
5.2 Ransomware Increasing [92] . . . . . 80
5.3 Threat Modeling [96] . . . . . 82
5.4 Mirai Botnet DDoS Attack [98] . . . . . 83
5.5 Carna Botnet DDoS Attack [99] . . . . . 84
5.6 Hardware and Software Supply-Chain Complexity [95] . . . . . 86
5.7 Source Code [37] . . . . . 88
5.8 Left: Control Flow. Right: Flattened [37] . . . . . 89
5.9 Compiler Phases . . . . . 93
5.10 CNN $\rightarrow$ Siamese Model . . . . . 102
5.11 DeepWalk Latent Representation [49] . . . . . 104
5.12 Random Walk [6] . . . . . 104

# CHAPTER 1

# INTRODUCTION

## 1.1 Security is easy to sacrifice

As demand for software increases, a common industry best practice has emerged: "don't reinvent the wheel". This saying is shorthand for a broader set of principles which are sometimes in tension. One of these is developer efficiency. If all other things are equal, increasing developer efficiency is desirable. Another principle is the performance of the software at runtime, from the perspective of the end-user or customer. If all else is equal, the user would prefer higher performing software. Security is another principle to round out this list. If all else is equal, both the business and their customers want software to be secure, not leak personal information, nor contain vulnerabilities that could ruin everyone's day.

This is by no means an exhaustive list of all the different priorities which must be balanced by software engineers, but it is an illustrative subset. These different aspects of a software artifact – performance, security, developer productivity – all exist in some proportions in any software project. Sometimes we find out about a security problem after a data leak or when a company decides to pay a ransom to regain control of their data.

From the sidelines, it is tempting to look down at such cases and imagine that we are more disciplined at writing software than those other folks. But the uncomfortable truth is that even highly talented and careful software engineers sometimes make mistakes. Sometimes there are hard deadlines or intense competition that encourages a shift in priorities, and often security is seen as the easy sacrifice.

Turning your focus away from security has short-term benefits but the long-term costs are unpredictable. As a simple example, many databases have a default administrator username and password such as admin/admin. Leaving these default settings makes it easier and faster to set up the database (short-term benefit). This is an easy target for a would-be attacker, which might

someday harm your company. But on the other hand, not every company who makes this mistake suffers the consequences. A principled approach to software security therefore requires not only constant vigilance to external threats, but also your own nature.

## 1.2 Costs and benefits of external dependencies

In this software development environment, when confronted with a need to implement a common feature, the best practice advice is to use a standard library if possible. If such functionality is not present in a standard library module, try to find a third-party module. Only if no such functionality exists in either the standard library or any third-party module, should a developer spend their own time to implement it. This ranked ordering of preferences has some positive consequences: module developers can afford to specialize on only the niche function covered by their module. By deferring responsibility of certain functions to specialists, most developers can be generalists. This is usually the right choice for an enterprise because it allows them to nimbly adjust to new products or market demands.

The downside is that as more enterprises rely on a small set of commonly used modules, those modules become attractive targets for attackers. From the attacker's point of view, it is more efficient to focus an attack on one module used by many companies than by targeting each individual company separately.

## 1.3 Make it easier to do the right thing

Given human nature and the pressure to compete, is security doomed to become a perpetual afterthought? Possibly not if we can lower the cost of doing the right thing. Currently, there are a handful of tools to help automate security analysis as part of the software development process. But generally these tools are either easy to use **or** they offer deep and valuable insights. This dissertation aims to find a more optimal balance between these two extremes.

## 1.4 Choose one part of the larger problem

These concepts of balancing security with features and performance are abstract and broad, and beyond the scope of a single dissertation. In the course of this research, the committee helped narrow the scope of research. Initially, we had ambitions to analyze the output of hardware synthesis tools used to configure field programmable gate arrays (FPGAs) and application specific integrated circuits (ASICs). These are forms of software-defined hardware, and we are concerned with the security of their respective supply chains. While this concern is justified, in order to implement the kernel of our idea (data augmentation) we discovered that these forms of code are far too limiting. Simply put, current FPGA and ASIC generation tools are far too slow to generate a sufficient quantity of synthetic data within any reasonable period of time.

## 1.5 Binary analysis

This dissertation focuses on the task of analyzing compiled binary programs. Unlike source code, compiled programs often lack familiar human-readable notation If the program has been stripped of debug symbols, then essentially all that is left is a sequence of fine-grained instructions without any contextual information about what the instructions should do or why they exist. The two main ways to analyze such programs are statically or dynamically. Dynamic analysis runs the code to observe its side-effects, while static analysis tries to assign probable meaning to the code without executing it. While dynamic analysis can demonstrably prove that a chunk of code has some effect, it is costly to run.

In general, dynamic analysis should be carried out in an isolated environment to protect the testing environment from either accidental bugs or malware. This isolation can take the form of a dedicated test machine or a virtual machine. Running a physical or virtual machine is slower than reading the code, and some sequences of instructions may never terminate, which means that exhaustive checking is impossible for dynamic analysis tools.

Static analysis on the other hand *can* include exhaustive checks because executable code has

finite length. Unfortunately, it does not replace dynamic analysis entirely, because the resolution of static analysis is limited to the meaning that it can assign to patterns of executable code. To a first approximation, dynamic analysis is slow but precise, and static analysis is fast but vague.

A less common hybrid between static and dynamic analysis is symbolic execution, in which part or all of the program's execution state is simulated with symbolic values rather than concrete values as would exist in a dynamic context. This approach can be used to explore some of the runtime execution paths more efficiently than dynamic analysis, but maintaining simulated states is costly.

### 1.6 Obfuscation as data augmentation

Binary analysis is made more difficult by intentional obfuscation in which the software developer makes changes to the syntax of the code in order to hide its meaning from static analysis tools. Stripped and obfuscated code is the most challenging to decipher. However, this dissertation uses one key insight about obfuscation to turn it into an advantage. Because obfuscation by design preserves the meaning of the original code, it can act as an effective form of data augmentation.

This allows us to synthesize larger data sets from a small set of original programs, and those data will retain the essential qualities of the originals despite being entirely different in appearance. We combine this shift in perspective with existing approaches to pattern detection from machine learning, and the result is a highly effective method of classifying unknown binary data. Unlike previous efforts in this domain, the approach in this work is robust to extreme levels of obfuscation. This scenario represents a worst-case failure mode of existing approaches, because a small increase in effort on the part of an attacker results in a complete failure of the defenses. Models trained to recognize obfuscation-augmented data stop the arms race between attackers and defenders by reducing the expected value of added obfuscation.

## 1.7 Contributions of this dissertation

In this document, we include three chapters exploring the obfuscation-augmented approach to binary analysis. The first of these chapters, *Data Augmentation*, explores different methods of data augmentation for binary code and motivates why obfuscation is more ideal than other methods. The next chapter, *Detecting Standard Library Functions in Obfuscated Code* applies this concept using an existing machine learning model to the task of identifying standard library functions. The third chapter, *Function Classification for Obfuscated Binary Code*, further refines this exploration with a comprehensive comparison of different types of machine learning models used across different representations of the same data. In addition, this third chapter explores the value added by inclusion of graph structural data of the binary code. A fourth chapter reviews the other scholarly works related to this topic and briefly touches on tangentially related topics such as electronic design automation tools. The benefits and limitations of our approach are summarized in a fifth chapter, and we summarize our conclusions in a final chapter before listing bibliographic references and curriculum vitae.

# CHAPTER 2

# DATA AUGMENTATION

A key challenge of applying machine learning techniques to binary data is the lack of a large corpus of labeled training data. One solution to the lack of real-world data is to create synthetic data from real data through augmentation. In this chapter, we demonstrate data augmentation techniques suitable for source code and compiled binary data. By augmenting existing data with semantically-similar sources, training set size is increased, and machine learning models better generalize to unseen data.

## 2.1 Introduction

Modern software development relies heavily on third-party code and open ecosystems, which aid developer productivity, but are also attractive targets for malicious actors. Attackers may release malware with a name similar to a real package in an attack called "typosquatting", or try to gain access to an existing project and quietly add malicious code[1]. New technologies such as GitHub's Copilot[1] provide developers with snippets of third party code sourced from public repositories, which in addition to aiding productivity, may also be used as an attack vector for injecting publicly available malware into new code.

Recent typosquatting attacks include a malware named `colourama`, which targeted the similarly-named Python package `colorama`. The malware bundled cryptocurrency mining code alongside the original code[2], so developers were less likely to notice the problem. Similarly, the node.js malware `crossenv` typosquatted `cross-env` and exfiltrated user data. Detecting and preventing typosquatting is an active area of research[3].

Today's developers need tools to help find hidden malware in third party code. Machine learning techniques are rapidly finding applications in this domain because models can be made general enough to classify never-before-seen code as either malware or not. However, a limiting factor for

[1] `https://github.com/features/copilot/`

machine learning model efficacy is the quantity of training data. One project used a large dataset of 66,388 PowerShell commands (6290 malicious)[4]. Another project sourced approximately 3500 implementations of 6 different algorithms from GitHub[5]. However, a single PowerShell command may represent a stand-alone program or a small part of a larger program, and not all projects have access to such a large volume of data as these examples.

When the quantity of available data is too small, a classic approach to artificially synthesizing more data is called *data augmentation*. In computer vision, image augmentation includes rotation, translation, adjusting color, or adding noise. In natural language processing, text is augmented by translating to another language and back again, adding or deleting words, and permuting word order. In this work, we examine data augmentation techniques suitable for source and compiled binary code analysis.

*Data Augmentation for Source Code.* Some data augmentation methods from the Natural Language Processing (NLP) domain also work on source code. Like natural language, source code can be thought of as a 1-dimensional sequence of symbols, with specific word formation rules. However, in natural languages it is rare for an author to invent new words, but programmers create new names routinely. Programming language is also more structured than natural language, with extensive use of explicit grouping structures like parentheses and semantically significant indentation and punctuation. These aspects make it harder to define "synonyms" for word-like tokens in programming languages or to perform transpositions without changing the program's meaning or breaking it completely.

Source code can also be thought of as a 2-dimensional image where each character or symbol represents a "pixel". This can be done by treating line feed characters as vertical offsets so the "image" is the same character grid as used by all common text editors. Sequential code can be embedded into a square matrix by treating sequences of two tokens as row/column indexes and recording occurrence counts of the two-token n-gram at each index.

Most code has non-linear control flow in the form of conditional branching, and can be embedded into a graph data structure as in [6]. This type of embedding can be further transformed into

a regular or rectangular shape such as an adjacency matrix which is more amenable to processing by machine learning libraries. These machine learning libraries typically expect samples to have uniform shape. Any of these interpretations can be further augmented by adding pseudo-random noise, for example additive Gaussian or Poisson noise as used by [7].

Data augmentation techniques which preserve the semantics of the original source code require specialized knowledge of the source code languages used. For C language, examples of tools which can alter the syntax while preserving semantics include obfuscator-llvm[8] and Tigress[9].

In this project, we apply code-specific data augmentation to standard library code and analyze how it affects the robustness of a machine learning model in detecting specific functions in compiled binary data.

### 2.2 Related Work

Recently, Bayer et. al [10] showed that data augmentation can be performed on natural language texts by leveraging a large pre-trained model like GPT-2. This allowed then to generate novel tokens for insertion into their data set. Critically, these novel tokens were derived from a large model which preserved the semantic relationships between words. This type of token generation would be beneficial to future work in code analysis, but first the field would need a large pre-trained model. One potential use of this technology could train a model on a large corpus of source code, and use this as a mechanism for generating summary tokens for local context. GitHub's Copilot project uses OpenAI's Codex language model[2], whose purpose is to translate natural language commentary into valid code. GitHub is well positioned to take advantage of the Codex model due to their massive collection of source code as well as commentary explicitly related to that code. While generating code from commentary is related to semantic analysis of obfuscated binary code, it is not identical. For example, in the task of reverse engineering an obfuscated binary blob, natural language commentary may be either non-existent or intentionally misleading as an additional form of obfuscation.

[2] `https://openai.com/blog/openai-codex/`

Marastoni et. al found that reshaping binary files into 2d image data allowed them to use conventional convolutional neural networks (CNNs) even though they arbitrarily restructured 1d data into 2d by choosing an image width of 64[11]. They showed that reshaping actual image data had negligible effect on accuracy, but reshaping compiled program data into different width images had a drastic effect on accuracy. Their explanation is that the model is more complex than simple digit recognition. Another consideration is that by imposing an arbitrary shape on one-dimensional data, their CNN finds correlations between instruction values that happen to be arranged on intervals of 64 bytes. This work uses Tigress to augment a selection of 47 C programs and ultimately obtains a distribution of 9400 programs representing the 47 original categories.

Gupta et. al applied a deep neural network with attention mechanism to find syntactic fixes to 6971 buggy student programs[12]. This task was able to make use of an oracle (compiler error messages) - if the compiler produced an error message, the program contained at least one bug. While this approach is language-agnostic, it is limited to only syntactic errors. Logical errors and malware cannot be detected by such an approach.

An early example of using data augmentation for malware detection is due to Catak et. al[7]. In this work, random noise is injected into examples of both regular and malware programs. Like [11], Catak et. al[7] use a CNN model after reshaping the programs into images. However, instead of using the raw byte data, they synthesize a color PNG image file, by substituting decimal conversion, entropy conversion, and zeros of the original program into the red, green, and blue pixel values, respectively. This extra step embeds more summary information into each pixel compared to [11].

Asm2Vec[13] builds on the foundation of Word2Vec[14] and applies the Paragraph Vector-Distributed Memory (PV-DM) concept [15] to assembly code. Like [11], this work also leverages Tigress and obfuscator-llvm for data augmentation. However, Tigress-transformed programs were omitted from the complete evaluation. A key method which distinguishes Asm2Vec is their use of control flow to embed a partial function call graph structure in addition to PV-DM of instructions and operands. Even though this type of graph embedding is not language agnostic, we believe control flow more accurately models the structure of the code and should be preferred to n-gram

embeddings or arbitrary reshaping into images.

### 2.3 Methods

Analysis of unknown binaries involves sorting through both familiar and unfamiliar code. Standard library code is included in many other programs, so the ability to robustly identify standard library functions aids security analysis and reverse engineering efforts. In this work, we analyze code from the `musl` C library[16]. It is used by projects such as Alpine Linux[3] and Emscripten[4]. The source code for `musl` in this work comes from the `musl-cross-make` toolchain at revision **b298706**[5]. We chose `musl` for this project because of its compactness and relative ease of compilation compared to similar implementations such as glibc[6].

Table 2.1: selected `musl` library functions

| Library | Function |
|---|---|
| ctype.h | isalnum |
| ctype.h | tolower |
| math.h | exp |
| math.h | floor |
| math.h | pow |
| stdio.h | fprintf |
| stdio.h | printf |
| stdlib.h | free |
| stdlib.h | malloc |
| stdlib.h | strtol |
| string.h | strcat |
| string.h | strstr |

The internal implementation of `musl` contains many examples of code reuse. This means that in order to compile a single `musl` function it is usually necessary to compile the library in its entirety. After compilation, the resulting object file contains several thousand public symbols. However, we restrict our analysis to a subset of library functions, given by Table 2.1. These functions are common to many programs, but more importantly this subset contains groups of

[3] `https://www.alpinelinux.org/`
[4] `https://emscripten.org/`
[5] `https://github.com/richard-vd/musl-cross-make/commit/b298706`
[6] `https://www.gnu.org/software/libc/`

similar functions. For example, `printf` and `fprintf` are similar, as are `exp` and `pow`, but `printf` and `pow` are different. Likewise, we expect functions from the same library to be more similar than functions from separate libraries. We use this similarity assumption as a baseline to indicate whether we have found reasonable clusters of related functions.

### 2.3.1 Data Augmentation by Source Code Obfuscation

Obfuscating source code means modifying the structure of the code without changing its core functionality. This can include trivial changes like renaming variables or more substantial changes like altering the structure with additional control flow, spurious functions, or even self-modifying code. For compiled languages, variable renaming has a negligible effect on the binary output, and even some structural changes like changing `while` loops to `for` loops may result in the same binary code after optimizations are applied. In this work, we obfuscate C sources using Tigress[17], a source-to-source transformer for C code. Tigress transformations include: splitting functions into smaller parts, merging multiple functions into one, flattening control flow, transforming functions into specialized interpreters, and many others. Each transformation results in a different source file, and multiple transformations may be combined to produce additional sources.

A challenge of using Tigress is that it can only transform a single C *program* file, not a *library* consisting of many files, so to use Tigress with `musl`, the recommended method is to merge separate files into one large C file. However, we determined this approach was infeasible, due to `musl-cross-make`'s complex multi-stage build process.

Instead, we use a workflow as shown in Figure 2.1. We begin with the `musl-cross-make` cross-compiler to build the shared object file `libc.so`, for the `x86_64-linux-musl` target. Then we *decompile* functions of interest (and their sub-functions), using Ghidra[18]. Decompilation is the process of generating C code from binary code. However, decompilation is an imperfect process, and Ghidra's decompiler does not always generate syntactically valid C code, so we manually fix the syntax errors in the decompiled C files. Through this method, we obtain a single-source representation of the source code for all function. All the functions are finally merged to form a

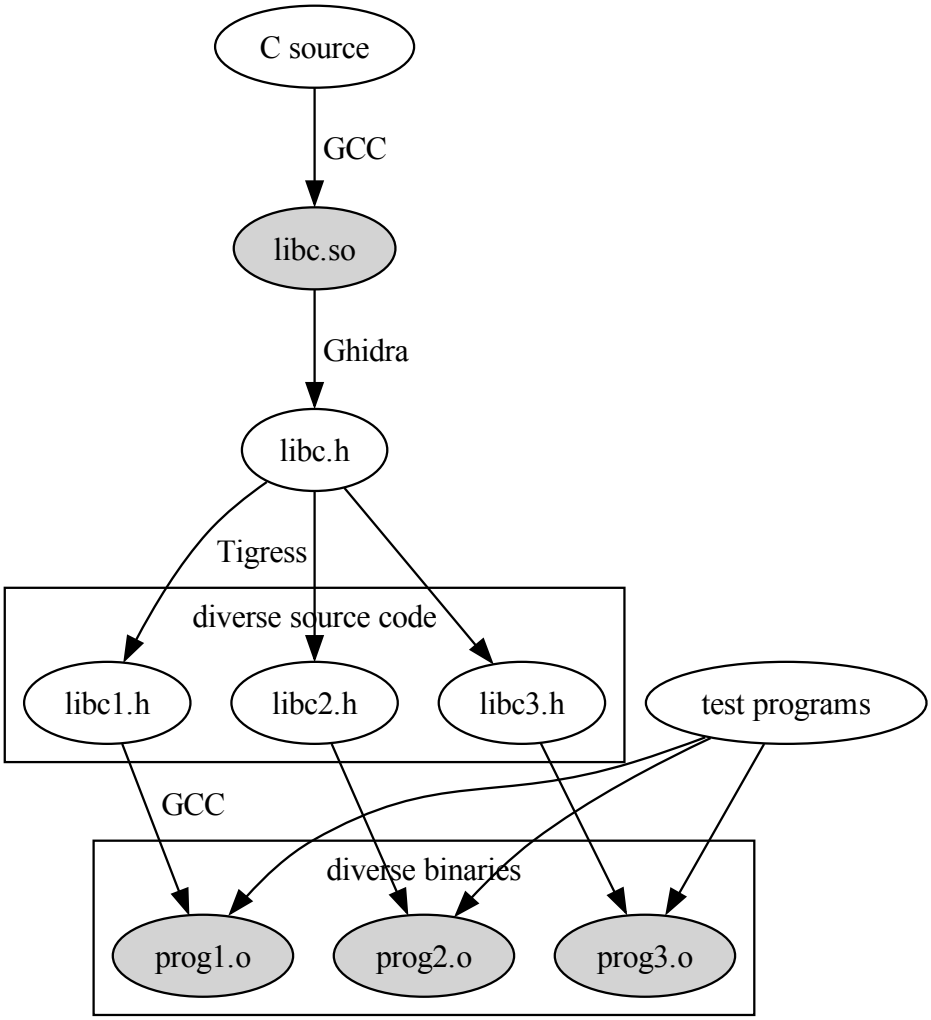



Figure 2.1: Generating diverse binaries from single sources

new library file, which we call **libc.h**.

### 2.3.2 Semantics-Preserving Transformations

Tigress offers many program transformations, and many of these transformations can be combined. We use two selected "recipes" from the Tigress website [7]: *Opaque Predicates, Branch Functions, and Encoded Arithmetic* and *Virtualization and Self-Modification*. In addition to these pre-made recipes, we also transform with *Encoded Arithmetic*, *Encode Branches*, and *Virtualize* separately, for a total of 5 separate transformed C sources. Generating 5 different sources is analogous to obtaining 5 different standard library implementations, in that they all have the same semantics, and only differ in implementation details.

### 2.3.3 Test Programs

Our approach aims to reliably detect standard library functions within an obfuscated binary. To test this, we wrote multiple test programs which call functions from **libc.h**.

We use a selection of trivial C programs as data classes. Each program contains one library function, and the filename describes the function under test, for example `pow.c` calls the `pow` function. These programs are short and trivial, averaging about 14 lines of code. The key feature of these programs is the inclusion of runtime data to prevent the compiler from optimizing away the function under test. For example, the file `pow.c` is shown in Listing 2.2.

```
int main(int argc,char**argv){
    return (int)pow(1.3, argc);
}
```

Figure 2.2: `pow.c` source code

Because the `pow` function uses the run-time parameter `argc`, its result cannot be computed at compile-time. This ensures the binary code for the `pow` function appears in the compiled output.

[7] `https://tigress.wtf/recipes.html`

Since all the test functions are trivial, the compiled outputs differ primarily by the library function they contain, plus a negligible amount of boilerplate code for the `main` function. A version of **libc.h** which provides the `pow` function is included at compile-time with `gcc`'s `-include` option. When we compile the code, we only compile (with `gcc -c`) and do not link, which means no link-time optimizations execute, and all **libc.h** functions appear in the object file, even if they are not used in the test program.

### 2.3.4 Data Augmentation Types

We consider data augmentation to be an important step to improve binary analysis techniques from a security perspective. Although there exist methods[19], [20] for determining binary semantics, they are sensitive to minor semantic changes - exactly the kind of changes that would be caused by the addition of malware patches to a binary. Other methods [21]–[23] rely on obtaining Control Flow Graphs (CFGs) from the binary or other types of "expert knowledge", which may induce bias. The main goal of using Data Augmentation is to improve model accuracy by increasing the amount of training data, all without introducing additional bias.

Data Augmentation involves making small changes to the source material which do not drastically change its meaning. For example, adding small amounts of Gaussian noise to a picture of a teapot results in a picture of a teapot with some bad pixels, not a picture of a cat. Similarly, image data is often augmented through linear transformations such as rotations, scaling, shear, and transposing along either vertical or horizontal axes. In NLP, sentences can be augmented without destroying their meaning in several ways - by replacing words with synonyms, transposing some words, or random insertions and deletions of words. In these examples, even though the transformations may affect a large portion of the data (e.g. every pixel), they do not cause *semantically* large changes to the data. However, directly applying the techniques of NLP to computer vision may not be appropriate - for example random rearrangements of training data pixels may not improve the accuracy of a computer vision system. Similarly, reversing sentence order is unlikely to improve an NLP system. Therefore, when we adapt machine learning techniques to the domain of binary

code analysis, we must consider what types of data augmentation are reasonable and will improve our model.

In this work, we compare two types of data augmentation: *semantics-preserving* and *NLP-style*. The *semantics-preserving* type is accomplished in two ways - first, using the Tigress obfuscator on the C source code, and second, by using multiple compilation options to produce multiple object files. The transformations provided by both Tigress and compilers such as GCC or Clang may affect performance, memory usage, or code size, but they should always preserve the meaning of the functions used (assuming no "undefined behavior" is invoked).

We define the *NLP-style* type to include permutations, additions, deletions, and substitutions of object code tokens. These augmentations are performed on disassembled text files prior to using them as training data in our model. This type of augmentation is more appropriate for binary code than true random noise, because random bit flips could cause the program to crash - and malware authors do not want their victim's programs to crash, but rather execute the malicious code without causing alarm.

An important note about the difference between *semantics-preserving* and *NLP-style* augmentations is that the *NLP-style* augmentations explicitly do not preserve the semantics of the original sources. While the resulting code can still be compiled and executed, it no longer performs the same function as the original code, and indeed may crash or corrupt data. We do not suggest that anyone attempts to execute code which has been randomly permuted in this way, because its semantics are undefined.

All of the code generated through the *semantics-preserving* transformations is always compiled, and if executed it retains the semantics of the original source code. We note that some of these obfuscations may increase memory usage, execution time, or both because obfuscation is in some ways antithetical to optimization. However, we define *semantics-preserving* broadly to include programs that have the same function but do not necessarily have identical running time or memory usage.

Finally, we note that the transformations we apply in this work are syntactically valid at every

phase. In the *semantics-preserving* phase, transformations occur at the level of C source code. This code is then compiled to binary then disassembled into assembly code. The assembly code is finally transformed by *NLP-style* transformations, which always maintain correct assembly language syntax.

### 2.3.5 Machine Learning Model

In this work we wish to focus on the impact that data augmentation has on a machine learning workflow. To that end we employ the established Asm2Vec[13] model. While we did spend some time optimizing hyperparameters for Asm2Vec, ultimately we kept most of the values the same as described by Ding et. al. Disassembly and training is facilitated by the open source implementation of Asm2Vec called Asm2Vec-pytorch[24]. The Asm2Vec model creates a vector embedding from textual assembly code, so we first preprocess the object files by disassembling them with the reverse engineering tool Radare2[25]. This step of the process transforms the compiled binary code into human-readable assembly code.

Asm2Vec creates an embedding for an assembly function by two methods: *graph* embedding and local *contextual* embedding. A function can be modeled as a graph of *basic blocks*, which are sequences of instructions delimited by `jump` or `call` type instructions. Jump and call instructions can be thought of as directional edges connecting one basic block to another. To obtain a graph embedding, some `call` and `jump` instructions are followed to their destination, resulting in a partial graph of a given function. Rather than exhaustively searching the complete basic block graph, Asm2Vec samples each list of outgoing edges with 3 random walks.

Contextual embedding is achieved by first considering a particular instruction as the current instruction, then examining the instructions immediately before and after the current instruction. The instructions before and after the current instruction are the "context" for the current instruction. These two instructions are tokenized into their opcodes and operands, and from these tokens an embedding vector is obtained by a method called *paragraph vector distributed memory* (PV-DM)[15], which represents the surrounding context for the current instruction.

The vector embeddings obtained by PV-DM and random walks are combined into a single long vector for each assembly function. Even though different assembly functions have different lengths and different numbers of basic blocks, the vectors generated by Asm2Vec are all 200 units in length. The uniform size facilitates more efficient training using modern hardware such as GPUs.

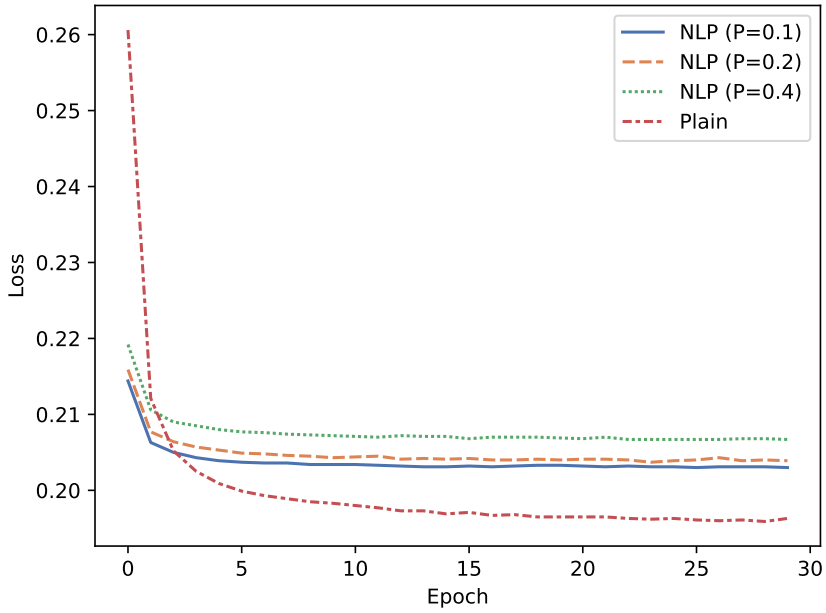



Figure 2.3: Training loss for data augmentation

To train this model, a corpus of assembly functions is used as input, and at each iteration the model attempts to predict the *current token* for each instruction in the function. When predicting the current token, a ranked list of candidate tokens is used. If the current token matches the highest ranked candidate, the gradients are unchanged. However, if the current token is in the list of candidates but not ranked highly, the gradients are changed. And finally if the current token is not contained in the list of candidates, the gradients are adjusted more. Predicting the current *token* is a finer grained task than predicting the current *function*.

Because Asm2Vec aims to find binary *clones* rather than *similar* binary functions, its loss function optimizes for predicting sequences of assembly tokens. In this work, we are more interested in *classification* of similar functions. Ding et. al [13] do not publish results for code which was

obfuscated by Tigress, due to the low accuracy of their model with this type of obfuscated data. We expand on Ding et. al by using their model on code which has been obfuscated by Tigress in multiple ways.

Figure 2.3 shows the mean of the binary cross-entropy between the predicted and actual tokens across the entire dataset at each epoch during training. This compares multiple loss profiles. Data marked *Plain* uses only the obfuscated disassembled functions, and data marked *NLP* adds copies of these obfuscated functions which are further transformed by a selection of methods inspired by NLP. Each line of text in the disassembled function contains either header information, labels, or opcodes/operands. The first 3 lines of the file contain header information, which we do not modify. We apply textual transformations to augment the existing assembly with additional variations. Those transformations include:

- delete lines
- duplicate lines
- transpose lines

For each line of assembly which is neither a header nor a label, we apply one of these transformations with probability $P = 0.1$, $P = 0.2$, or $P = 0.4$.

As figure 2.3 shows, data containing any *NLP*-augmented assembly converges more quickly but to a higher loss value than the *Plain* assembly code. Loss increases to $\approx 0.21$ as the probability of a transformation increases. Data containing only *Plain* assembly converges more slowly to a loss of $\approx 0.19$.

Table 2.2 shows excerpts from two disassembled object files, each based on `fprintf`, with the options `-O2` and `-funroll-all-loops` added during compilation. The two columns show the *original* disassembled file on the left, and on the right, a version of that file which has been augmented with *transposed* lines, highlighted in **bold face**. The transposing occurred in this example with probability $P = 0.1$. Lines 7 and 26 of the original replace lines 20 and 28 of the transposed version, respectively.

Table 2.2: Assembly augmentation by transposing lines

| line | original | transposed |
|---|---|---|
| 1 | .name sym.__fwritex | .name sym.__fwritex |
| 2 | .offset 000000000800b9b0 | .offset 000000000800b9b0 |
| 3 | .file fprintf-O2-unroll-all-loops.o | .file fprintf-O2-unroll-all-loops.o |
| 4 | LABEL102: | LABEL102: |
| 5 | cmp r9, rbp | cmp r9, rbp |
| 6 | cjmp LABEL11 | cjmp LABEL11 |
| 7 | **mov eax, dword [rbx+CONST]** | mov eax, dword [rbx+CONST] |
| 8 | test eax, eax | test eax, eax |
| 9 | cjmp LABEL14 | cjmp LABEL14 |
| 10 | mov r11, rbp | mov r11, rbp |
| 11 | mov rax, rbp | mov rax, rbp |
| 12 | and r11d, CONST | and r11d, CONST |
| 13 | LABEL29: | LABEL29: |
| 14 | sub rax, CONST | sub rax, CONST |
| 15 | cmp byte [r8+rax], CONST | cmp byte [r8+rax], CONST |
| 16 | cjmp LABEL11 | cjmp LABEL11 |
| 17 | cjmp LABEL18 | cjmp LABEL18 |
| 18 | lea rax, [rbp+CONST] | lea rax, [rbp+CONST] |
| 19 | cmp byte [r8+rax], CONST | cmp byte [r8+rax], CONST |
| 20 | cjmp LABEL11 | **mov eax, dword [rbx+CONST]** |
| 21 | cmp r11, CONST | cmp r11, CONST |
| 22 | cjmp LABEL18 | cjmp LABEL18 |
| 23 | cmp r11, CONST | cmp r11, CONST |
| 24 | cjmp LABEL25 | cjmp LABEL25 |
| 25 | cmp r11, CONST | cmp r11, CONST |
| 26 | **cjmp LABEL27** | cjmp LABEL27 |
| 27 | cmp r11, CONST | cmp r11, CONST |
| 28 | cjmp LABEL29 | **cjmp LABEL27** |
| 29 | cmp r11, CONST | cmp r11, CONST |

Similarly, for each assembly file, we also use *random deletion* and *random duplication* of lines. In each of these transformations, we leave the first 3 lines of header, as well as `LABEL:` lines unaltered. Interestingly, using data augmented with NLP-inspired techniques makes the training loss worse, which we discuss in the Results section of this work.

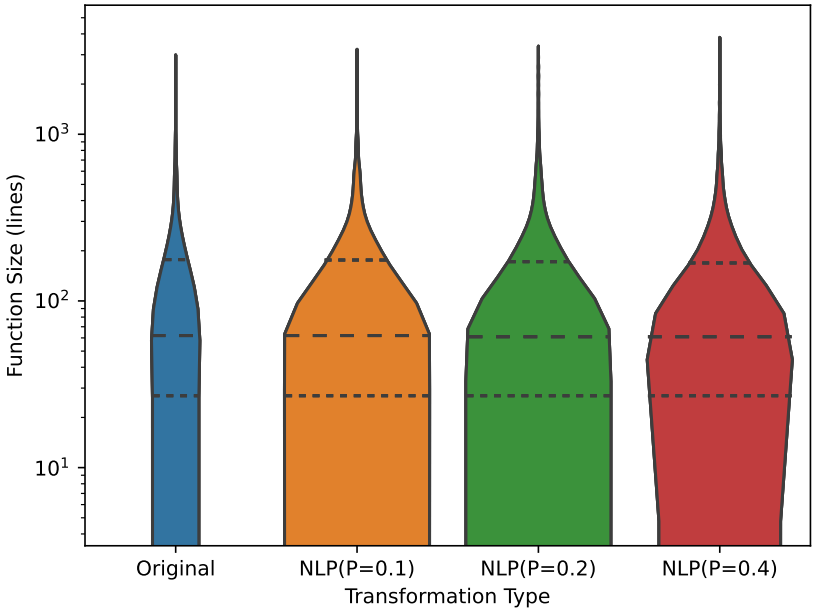



Figure 2.4: NLP-style augmentation maintains function size distribution

For each assembly file, we end up with a total of 4 copies: the original file, as well as *random deletion*, *random duplication* and *random transpose* variants. This means that after these NLP-inspired transformations, the data set could be biased to favor larger assembly language functions. However, while the shortest assembly functions are only a few lines long, which after the *random deletion* transformation can indeed reduce their size by a non-negligible amount, the number of such short functions is small. The width of each sub-plot in Figure 2.4 corresponds to the number of samples with the number of lines in the file given by the y-axis. The distribution of function sizes is unchanged by NLP-style augmentation, indicating the data set is not biased in terms of function size.

## 2.4 Results

Starting with a single `libc.h` source file, we apply 5 different transformations with Tigress to obtain 5 obfuscated versions of `libc.h`. Next, we include this obfuscated source in each of 14 different test programs (one for each test function). Each of these programs is compiled with diverse options:

- 5 optimization levels: O0, O1, O2, O3, Os
- 5 loop optimizations: unroll-loops, unroll-all-loops, unswitch-loops, loop-optimize, strength-reduce
- 2 C standard versions: c99, gnu99

All of these different options are combined in a Cartesian product as shown in Algorithm 1, for a total of $5 \times 14 \times 5 \times 5 \times 2 \times 2 = 3500$ different permutations from one source. An important consideration when performing this type of data augmentation is that any bias present in the original data will not necessarily be mitigated by the augmentation. This is analogous to image augmentation for computer vision tasks - if the data set contains only images of cats, no amount of augmentation will help the model recognize dogs. Likewise when augmenting source code via obfuscation and diverse compilation, the semantics of the original source will be replicated in the augmented versions.

**Algorithm 1** Diverse compilation

```
for libc in headers do
    for func in functions do
        for opt in levels do
            for loop in optimizations do
                CC -opt -loop -std=std -include libc func.c
            end for
        end for
    end for
end for
```

This method of adding diversity through both obfuscation and diverse compilation options is ex-

tremely valuable for this type of source code analysis, because it is difficult to obtain enough different implementations of the source material to satisfy the requirements of a machine learning approach.

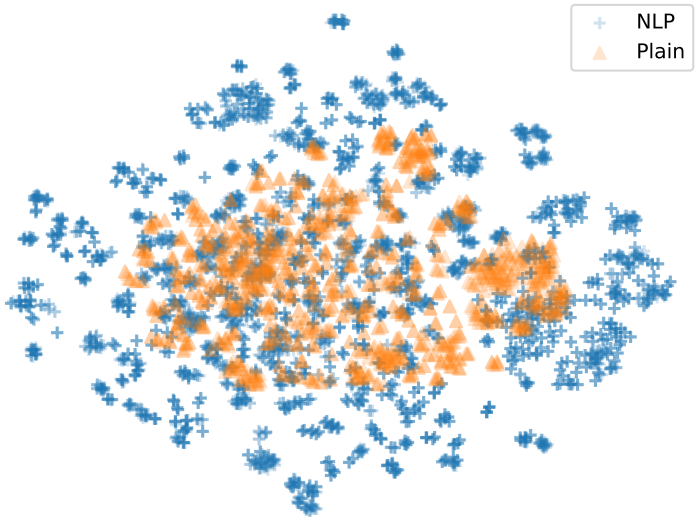



Figure 2.5: t-SNE of *Plain* and *NLP*-augmented data

As shown by Figure 2.5, adding NLP-style augmentation improves clustering performance when used with t-distributed Stochastic Neighbor Embedding (t-SNE). Part of this is simply because the NLP dataset contains 4x more samples (6044 versus 1511). However, the addition of NLP augmentation increases the model's generality and also allows for better discrimination of samples. Figure 2.6 shows the t-SNE clustering of the augmented data. There are some clearly visible clusters, especially for the `free` function.

We compare disassembled function embeddings using cosine similarity, in the same manner as Ding et. al in their Asm2Vec work[13]. A major difference between our work and [13] is due to our use of Tigress for obfuscating the C source code. Asm2Vec did apply obfuscation, but it was performed on the intermediate representation of compiled code only, using obfuscator-llvm [26]. This work not only obfuscates the C source code, but further transforms the generated assembly

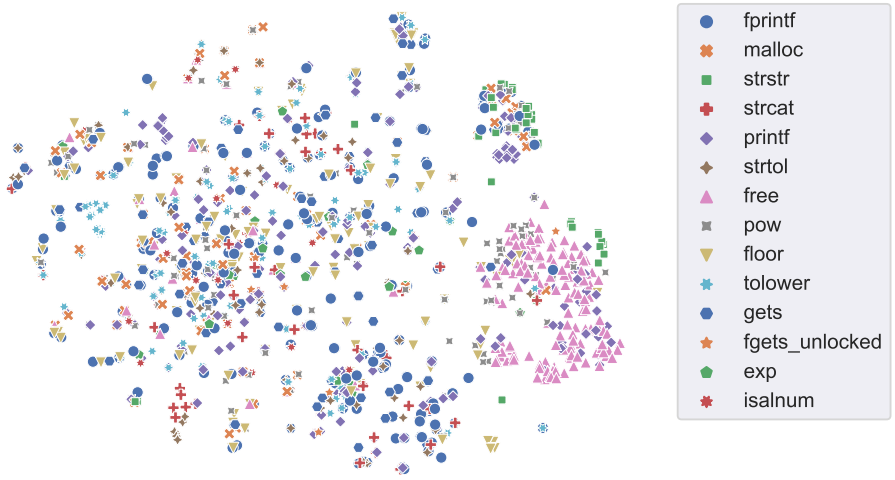



Figure 2.6: t-SNE, labeled

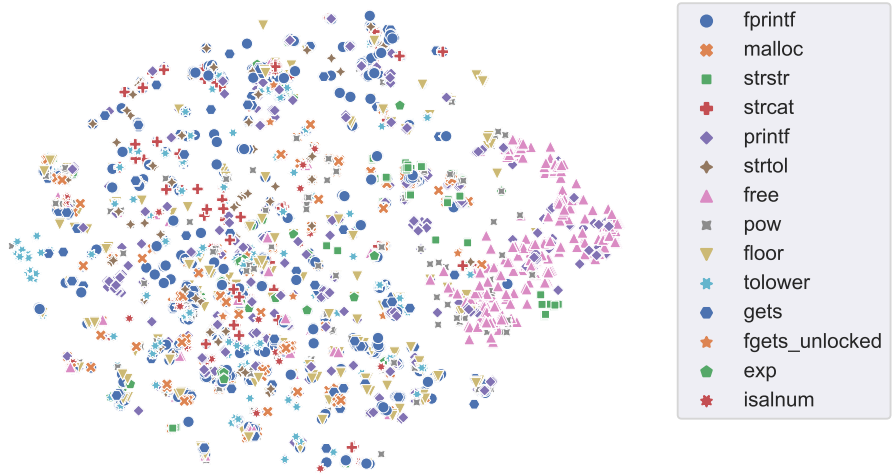



Figure 2.7: t-SNE of NLP augmented data

code using NLP-inspired techniques.

Generating diverse disassembly results in 6044 assembly files. Because each pairwise comparison takes about 1 second on an 18-core x64 workstation, exhaustively comparing all (6044× 6044) pairs would take over a year. Instead, we first generate all possible pairs and then sample with probability $P = 1/500$. Finally, we take the average similarity of each pairing to generate Figure 2.8. Some functions are more consistently similar, such as `pow` and `exp` or `printf` and `fprintf`. The fact that the diagonal of this heatmap does not have consistently high values may be attributed the low sampling rate. For example when row and column are both `fgets` or both `free`, we expect the similarity to be very high but it is only moderately higher than average. The `strtol` function is consistently dissimilar with all other functions.

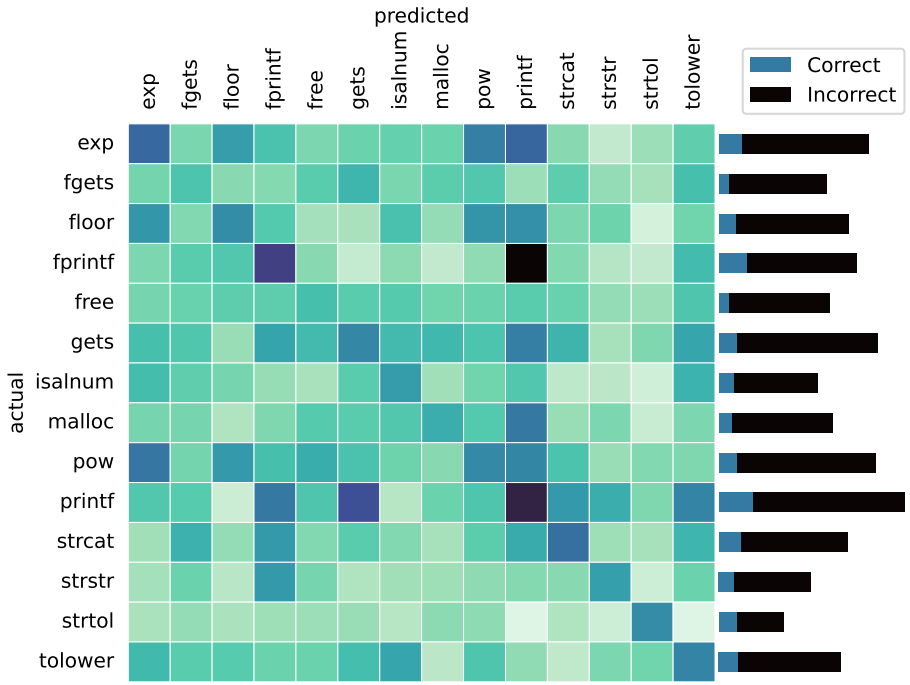



Figure 2.8: Mean function similarity

## 2.5 Conclusions and Future Work

This work demonstrates two contributions to the field of binary security analysis. First, we describe a method of generating a large amount of data from a single binary source. Second, we show a

novel application of data augmentation for binary code.

Word deletion, insertion, and transpositions are data augmentation techniques which previously were used mostly for NLP. One possible enhancement of this work is to leverage models which were trained on larger data, but in different domains. This method of *transfer learning* has proven effective in NLP and other domains, where a model trained on a large data set can then be used with good results on a smaller data set. The challenge for us in using transfer learning would be to determine what existing large models are a good match for code analysis. As noted by Ding et. al[13], natural language data is typically a 1-dimensional stream of tokens, whereas code contains multiple 1-dimensional streams which are explicitly linked to form a graph. As noted by Bayer et. al[10] a potential improvement to this work would leverage a larger pre-trained model. However, a significant challenge of applying a pre-trained model is that most large models are trained on natural language data rather than code, so it is unclear if this would be good or bad for code analysis.

Existing state-of-the-art models in this domain primarily focus on binary clone detection [11], [19], [27], [28]. These applications focus on finding exact or nearly-exact matches in order to prove cases of intellectual property theft or to detect known malware. In the face of obfuscation, the problem becomes much more difficult, because not only are exact binary matches much less likely to occur, but approximate matches may also be rare. This work aims to quantify which types of data augmentation are valid for code. One of our future goals is to build on this initial work and determine which types of data augmentation work best for obfuscated code.

By treating disassembled tokens as words, NLP-based techniques can be applied to any binary code for which a compatible disassembler exists (i.e. all major architectures). We find that NLP-based augmentation of the disassembled code improves clustering performance for t-SNE, but has worse loss characteristics during training.

This work examined only object files compiled for the x86_64 instruction set architecture (ISA). Future work should determine how well these results transfer to other ISAs, especially ARM and MIPS, although we anticipate that the technique should remain effective regardless of the ISA.

Additional research is warranted to determine what types of NLP-based data augmentation works best for assembly code. In this work, the types of changes we made were line-based, and finer-grained changes within lines could also change operands in addition to opcodes.

Finally, this work uses only one type of machine learning model (Asm2Vec). Other NLP-based models are likely also a good fit for the type of disassembly data we use in this work. We also use only one set of Asm2Vec hyperparameters. This could be further expanded and optimized by hyperparameter tuning to determine whether longer random walks (used for embedding) are more beneficial. Additional graph embeddings are possible using the generalized jump and call instructions in the disassembly code, and we plan to explore these types of embeddings and Graph Neural Network applications in future works.

# CHAPTER 3

# DETECTING STANDARD LIBRARY FUNCTIONS IN OBFUSCATED CODE

Binary analysis helps find low-level system bugs in embedded systems, middleware, and Internet of Things (IoT) devices. However, obfuscation makes static analysis more challenging. In this chapter, we use machine learning to detect standard library functions in compiled code which has been heavily obfuscated. First we create a C library function dataset augmented by obfuscation and diverse compiler options. We then train an ensemble of Paragraph Vector-Distributed Memory (PV-DM) models on this dataset, and combine their predictions with simple majority voting. Although the average accuracy of individual PV-DM classifiers is 68\Finally, we train a separate model on the graph structure of the disassembled data. This graph classifier is 64% accurate on its own, but does not improve accuracy when added to the ensemble. Unlike previous work, our approach works even with heavy obfuscation, an advantage we attribute to increased diversity of our training data and increased capacity of our ensemble model.

## 3.1 Introduction

Unlike dynamically linked applications which share library code with other applications, statically linked binaries bring their own libraries. Static linking has multiple advantages. It makes applications convenient for users to install. It also gives software developers full control over library code used by their application, and allows them to include patched libraries for custom extensions, or older versions of libraries for backwards compatibility. However, this flexibility also means security researchers must examine statically linked library code once per *application*, whereas code shared by dynamically linked applications only needs to be analyzed once per *system*.

Binary analysis techniques can detect malware in either dynamically or statically linked binary files [29], [30], but malware authors consistently try to thwart such analysis. One method of hiding malware is to strip[1] symbols and replace meaningful names with less meaningful symbols or

[1] `https://linux.die.net/man/1/strip`

numbers after compilation. Another method involves obfuscating source code before compilation. Extensions to mainstream compilers like LLVM [31] provide obfuscation as a compilation option [26], which makes distributing an obfuscated version of a binary no more difficult than distributing one without obfuscation.

After years of progress by reverse engineers and security researchers, obfuscation is still a challenge for modern static analysis [32]–[34], and some forms are "provably hard for any static code analyzer to overcome" [35]. Yet despite its shortcomings, static analysis remains valuable in part due to its low cost to implement. Because it never executes potentially malicious code, static analysis requires no special run-time environment. This contrasts with dynamic [36] or hybrid [37] analysis types which require isolated test environments and are therefore more costly.

In this work, we propose a static analysis approach to classify known functions in obfuscated and stripped binaries. Our method uses a general purpose machine learning model on a dataset prepared using domain-specific knowledge and tools, and attains accuracy up to 74%. Meanwhile, previous state of the art approaches like Asm2Vec [13] perform well on binary code whose sources were not obfuscated, but poorly when sources were heavily obfuscated.

**Research Summary:** Our method is robust to source obfuscation, which represents an improvement over previous approaches. To demonstrate this technique, we classify C standard library functions. Even though such functions are typically compact in their source code representation, they appear frequently in real programs, and IDA Pro's FLIRT [38] plugin demonstrates that there is commercial demand for this task. Unfortunately, malware authors can circumvent analysis by spoofing FLIRT signatures [39], which limits the utility of FLIRT as an analysis tool.

Our model-based approach is not susceptible to this type of spoofing attack because it does not rely on precisely matching signatures of function data, representing an improvement which could be useful in commercial security analysis products. The goal of this research is to improve the effectiveness of static binary analysis by reliably detecting known functions within an unknown binary. This lets security researchers spend more time dissecting *unknown* functions. Ultimately, our goal with this work is to improve the speed and quality of the practice of binary analysis.

The remaining sections of this paper are as follows: First, a Related Work section summarizes the state of the art in this domain and outlines some limitations of those approaches. Next, we describe in a Methods section the details of our approach. A subsequent Discussion section provides detail on our experimental approach and its limitations. After this, a Results section illustrates how our approach fares in our experiments. Finally, we detail our Conclusions and Future Work to reflect on the merits of our approach and what enhancements may exist for this approach.

### 3.2 Related Work

An important inspiration for our work is a technique called asm2vec [13]. The Asm2Vec model takes inspiration from **Word2Vec** [14], which in turn uses a Paragraph Vector - Distributed Memory (PV-DM) technique [15]. PV-DM is usually applied to natural language processing (NLP), but asm2vec shows it is feasible to treat assembly code as "words" for such a model. Like asm2vec and **InnerEye** [40], we disassemble the binary and tokenize the resulting assembly code, but our preprocessing step preserves the distinct labels. However, current related work either is not specifically focused on finding library functions, or did not consider obfuscated source material. We use an obfuscated training corpus, which allows our model to classify heavily obfuscated functions that were impossible for approaches like asm2vec. Our approach attempts to fill the gap in the current research by focusing on library function detection while tolerating a high degree of obfuscation. In addition, instead of discriminating between malware and non-malware, our approach aims to classify individual functions. Table 3.1 summarizes some of the state-of-the-art approaches in relation to ours. In this table, the term *Obfuscation Tolerance* means the method performed to a high standard on obfuscated code.

Recently, Yu et. al [43] explore augmenting source code by semantics-preserving transforms. However, their work explicitly tries to preserve the meaning and readability of the source code, not obfuscate it. In [45], Mi et. al use transformations such as variable renaming and modifying commentary to produce augmented variants of source code in order to classify the code as *readable* or not. Modifying source code comments has no affect on its run-time semantics, but renaming

Table 3.1: Comparison of Current Approaches

| Method | Purpose | Obfuscation Tolerance |
|---|---|---|
| Asm2Vec [13] | malware detection, function similarity | limited |
| InnerEye [40] | cross architecture basic block similarity | unknown |
| E Unibus Pluram [41] | defensive obfuscation | yes |
| AlphaDiff [28] | cross-version similarity | no |
| Structure2Vec [42] | scalable data representation | unknown |
| Catak et. al [7] | malware detection | unknown |
| Yu et. al [43] | data augmentation | unknown |
| Marastoni et. al [11] | binary similarity | yes |
| FLIRT [38] | library function detection | no |
| Qiu et. al [44] | library function detection | unknown |
| Ours | library function detection | yes |

variables would be considered a trivial or minor form of obfuscation.

Franz et. al suggest diverse compilation as a defensive technique in their 2010 paper [41]. They propose generating semantically equivalent but syntactically different versions of a program such that each user receives a unique version of the same program, thus limiting the scope of any specific attack or vulnerability exploit.

Because malware detection is a common goal in this domain, researchers commonly employ a Siamese network [28], [42] to train a loss function to determine which files are malware. While effective for classifying a given binary file as either malicious or benign, this approach is less suitable for recognizing specific subroutines or functions.

Data augmentation is often used for text classification as in [10]. Catak et. al [7] used data augmentation for malware detection, by injecting random noise into their data samples. Marastoni et. al reshaped 1-dimensional binary files into two-dimensional images, in order to use Convolution Neural Network (CNN) techniques [11] originally developed for computer vision applications.

Qiu et. al define library functions as those whose "instruction sequence and semantics are known". They also point out the value in identifying these types of functions because analyzing them would be a waste of time [44]. Their technique relies on constructing what they term an "execution dependence graph" in order to capture the semantics of the analyzed code.

Finally, graph reduction techniques have seen recent use [46] to train graph neural networks more

efficiently. This approach shrinks the overall size of the graphs while retaining some of their characteristics, such as relative ordering of nodes or average out-degree. Alternative graph reduction techniques exist such as [47] which preserve spectral properties of the graph. These reduction techniques have the potential to enhance our work by enabling faster comparisons on simpler graphs. Embedding techniques such as GraphSAGE [48] allow more efficient low-dimensional embeddings of large amounts of graph data. This technique may prove useful as our data sizes increase. DeepWalk [49] is a method for embedding graph data in a lower-dimensional vector space that is particularly effective when labeled data is sparse. Our use case also deals with sparse labeled data, so this type of embedding method is potentially useful when preparing our dataset.

### 3.3 Methods

Data augmentation techniques from computer vision and natural language processing do not necessarily apply to binary code, because even small changes to the code can result in unrelated semantics or invalid code. By using *obfuscation* and diverse *compiler options*, we can generate alternative versions of a given function without dramatically changing its semantics. We use the obfuscation tool **Tigress** [17], [50] and the **gcc** compiler[2]. Initially we select at random a set of functions from the `musl` C standard library[3]. However, some newer C language features used by the `musl` authors are not supported by the older C parser used by Tigress.

Table 3.2: Functions of interest (compatible with Tigress)

| | | | | | | |
|---|---|---|---|---|---|---|
| abs | acos | asin | atan2 | ceil | cos | daemon |
| exp | floor | fread | inet_addr | inet_aton | isalnum | kill |
| memccpy | memcmp | memmem | readdir | signal | sin | stpcpy |
| stpncpy | strchr | strcpy | strncpy | strstr | strtok | tan |
| vfprintf | wait4 | waitpid | | | | |

Table 3.2 shows the 31 remaining compatible functions. We then derive multiple new C sources from each function using Tigress and different compiler options.

[2] `https://gcc.gnu.org/`
[3] `https://musl.libc.org`

### 3.3.1 Data Augmentation Through Obfuscation

For this work, we use Tigress to augment both the size and diversity of our dataset. Given the source code for a C library function such as `abs` (Figure 3.1), we transform the source code file into approximately 345 new source code files, each based on different combinations of Tigress transformations and compiler options.

```
int abs(int a) {
    return a>0 ? a : -a;
}
```

Figure 3.1: C source code for `abs` function

Figure 3.2 compares and contrasts different some of the types of obfuscation. First we show the normalized assembly code for `abs` without obfuscation. Next is `abs` obfuscated with the `Split` transformation, which splits a single function into multiple parts. Finally we show `abs` obfuscated with both `EncodeArithmetic` and `Flatten` transformations. This demonstrates the diversity provided by obfuscation, allowing even a short function like `abs` to be effectively augmented.

```
1  endbr64
2  mov  eax, edi
3  neg  eax
4  cmovs  eax, edi
5  ret
```

No obfuscation

```
1   endbr64
2   mov  eax, edi
3   cdq
4   mov  ecx, edx
5   xor  ecx, edi
6   sub  edx, ecx
7   xor  edx, edi
8   cjmp  LABEL7
9   neg  eax
10  LABEL7:
11  ret
```

EncodeArithmetic, Flatten

```
1   endbr64
2   sub rsp, CONST
3   mov dword [rsp + CONST], edi
4   test edi, edi
5   cjmp LABEL4
6   lea rsi, [rsp + CONST]
7   lea rdi, [rsp + CONST]
8   call LABEL7
9   LABEL7:
10  jmp LABEL8
11  LABEL4:
12  mov eax, edi
13  neg eax
14  mov dword [rsp + CONST], eax
15  LABEL8:
16  move eax, dword [rsp + CONST]
17  add rsp, CONST
18  ret
```

Split

Figure 3.2: Selected `abs` assembly code variations

Because Tigress is a source-to-source transformer, it permits sequential transformations for even greater diversity. Some obfuscation types are more relevant for intentional obfuscation than oth-

ers, particularly those which are proven to be NP-complete such as the control flow alterations described by [51]. One malicious obfuscation technique hides itself from IDA Pro's FLIRT function detector [39]. In this attack, malware authors construct malicious code which matches one of FLIRT's library function signatures, causing FLIRT to mark the malicious code as benign. The existence of this attack demonstrates a vulnerability of signature-based static analysis. Basic obfuscation such as encoding literal values or inserting no-op codes are easier for traditional static analysis than more advanced obfuscation which alters control flow.

All Tigress transformations we use preserve enough of the original semantics such that the code compiles without errors or warnings. However, the resulting binary file may not exactly match the original semantics, especially in terms of performance. Despite these slight differences, obfuscation is a more realistic form of data augmentation than inserting random noise into the binary file as in [7]. We choose combinations of the following Tigress transformations[4]: (Flatten, Split, EncodeArithmetic, Virtualize, AntiAliasAnalysis, EncodeLiterals, InitEntropy, InitOpaque). These transformations cover a spectrum from "basic" to "advanced" obfuscation types. EncodeArithmetic and EncodeLiterals are "basic" transforms which do not disrupt control flow. Split, Flatten, and Virtualize are more "advanced" transforms that do affect control flow and are traditionally pose a challenge for static analysis.

Our work uses each of these transformations individually to produce one set of outputs, and then produces further outputs by forming sequences of up to three transforms. Some example sequences of transforms include:

- (EncodeLiterals → EncodeLiterals → Flatten)
- (Flatten → Virtualize → AntiAliasAnalysis)
- (Virtualize → Flatten)
- (Flatten → AntiAliasAnalysis → Flatten)
- (Split → Split → Split)

[4] `https://tigress.wtf/transformations.html`

In addition to obfuscation, we also use *compiler options* to add diversity. We keep all compiler options originally used by `musl`, and add more of our own to obtain additional binary files. We collect groups of these options and apply an entire group during compilation:

**Loops** -floop-parallelize-all, -ftree-loop-if-convert, -funroll-all-loops, -fsplit-loops, -funswitch-loops

**Codegen** -fstack-reuseall, -ftrapv, -fpcc-struct-return, -fcommon, -fpic, -fpie

**Safety** -fsanitizeaddress, -fsanitizepointer-compare, -fsanitizeundefined, -fsanitize-address-use-after-scope

**Optimize** -O3 (overrides -02 default value)

Each Tigress transformation is then compiled with one of these option groups at most, resulting in permutations of Tigress transforms and option group values such as (AntiAliasAnalysis → Virtualize → Flatten → **Loops**) and (Virtualize → EncodeArithmetic → Split → **Optimize**). This results in 1290 different combinations of transformations and options, 358 of which result in syntactically valid C code for at least some of our functions. We then discard any C code generated by Tigress which fails to compile. Finally, we apply each combination to each of our functions of interest, resulting in a total training set size of **9414** object files.

### 3.3.2 Data Preprocessing

For each compiled object file, we must transform it into a format usable by our models. This entails disassembling the object file into assembly code. This assembly code is further normalized by removing header information, replacing numeric values with a single token (`CONST`), and renaming all numeric offsets as generic labels, such as `LABEL5` or `LABEL194`. These two steps serve to *strip* the code of any identifying information, similar to using a linker's "strip symbols" option or the `strip` command [1].

The sections punctuated by `LABEL:` tokens are *basic blocks*, essentially linear sequences of instructions. If we again refer to the `Split` assembly code from Figure 3.2, we can describe its

graph structure as in Figure 3.3.

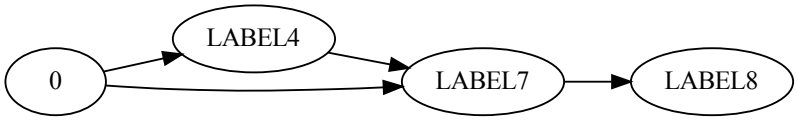



Figure 3.3: Graph representation of `Split`

The majority of this normalization step is provided by an open source implementation of asm2vec available from GitHub [24]. This assembly representation has replaced numeric offsets with `LABEL<N>` keywords, where N is a relative offset within the given file. There are also 3 lines of header at the beginning (`.name sym.abs`, `.offset 00000000800004e`, `.file abs.o`) in the original assembly output file. These lines identify the function explicitly, so we remove them from both our training and testing datasets, and they are not shown in the examples used here for clarity. Finally, we separate all tokens by whitespace and remove newlines, so the resulting document is a single line, part of which is shown in Figure 3.4.

```
endbr64 sub rsp , CONST mov dword [rsp + CONST] , edi test
```

Figure 3.4: Normalized assembly tokens

Each assembly file is now a single line of space-separated tokens, which we then combine into one multi-line *corpus* text file. This file contains neither function identifiers nor absolute numeric offsets, so we consider it equivalent to a stripped binary file. The corpus used in this work encodes **9414** documents in just **31MB** - comparable in size to approximately 10 photos from a high resolution smartphone camera. We mention the size of this dataset to illustrate that even this uncompressed plain text encoding is "small data" by today's standards, and this suggests that much larger training data sizes (1-2 orders of magnitude larger) are feasible for future work without requiring any change to the approach.

To measure the performance of our unsupervised model, we also store the name (label) of each function in a separate file, with one function name per line. The line containing the function name is the same line at which the corresponding function's tokens appear in the corpus. We later use this correspondence to validate the accuracy of the model, but because this is an unsupervised model, at no point in training or testing is the model exposed to these labels.

### 3.3.3 PV-DM Voting Classifier

We use Gensim's "Doc2Vec" implementation of PV-DM [5]. PV-DM contrasts with the Distributed Bag of Words (DBOW) approach by accounting for the order in which the words occur. This model aims to maximize the average log probability that word $w_t$ appears in a sequence of training words $w_1, w_2, ..., w_T$ within a window of size $T$ using the objective function in Equation 3.1:

$$\frac{1}{T}\sum_{t=k}^{T-k} log\ P(w_t|w_{t-k}, ..., w_{t+k}) \tag{3.1}$$

The hyperparameters for this PV-DM model are: **embedding size**: 400, **window size**: 10, **min count**: 10, **epochs**: 40. These are approximately the same as those recommended by [15]. However, we use 40 epochs instead of the recommended 10 due to our smaller training dataset. After training this model using a shuffled 10-fold cross validation split, discrete assembly language tokens are *embedded* into a lower dimensional vector space.

To visualize this embedding, we plot the first two principal components as shown in Figure 3.5. We annotate a subset with the token's name, to illustrate which tokens occupy similar regions of the embedding. Tokens in the `LABEL` category are not annotated, to avoid cluttering the visual. The most interesting aspect of this visualization is that labels (in orange) cluster near one another.

### 3.3.4 Graph Metadata Classifier

While the PV-DM model alone is capable of approximating some assembly language semantics, assembly code also contains *graph* structure, which models might use to more accurately classify

[5] `https://radimrehurek.com/gensim/models/doc2vec.html`

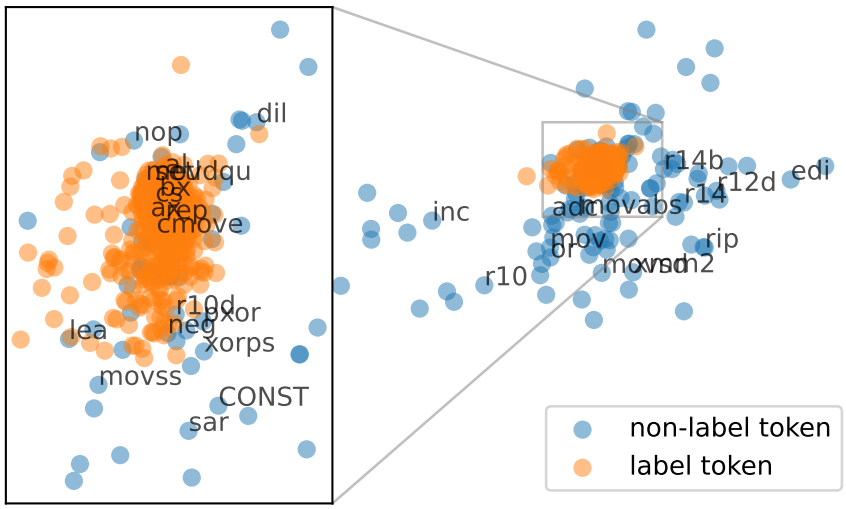



Figure 3.5: First 2 principal components of **Word2Vec** embedding (with labels)

functions. This graph structure is due instructions like `jump` or `call` making directed edges which terminate at `labels`. We partition sequences of tokens into nodes by splitting at every `LABEL<N>:` type token. This simplistic partitioning is coarser than a control flow graph because it ignores conditional jumps. Some types of obfuscation, such as Tigress' `EncodeLiterals` transformation, have minimal or no effect on this graph structure. Others, such as `Flatten` and `Split`, explicitly add or remove control flow, resulting in changes to the graph structure.

We reduce the complexity of these graphs using standard graph algorithms: **graph condensation** and **minimum spanning tree**, with implementations provided by the **graph-tool** software[6]. Graph condensation combines nodes if they form a strongly connected component and allows us to prune duplicate edges. For obfuscated code graphs, reducing the complexity in this way may approximate the shape of graph of the function before obfuscation. In practice, the reduced graph is not necessarily isomorphic to the original, and because these graphs only represent the *structure* of the code, they can not be checked for validity.

Figure 3.6 shows the effect of reducing an obfuscated graph using standard graph techniques. In each sub-figure, we show an example graph with node sizes weighted by the number of tokens present in the original basic block, and edge thickness proportional to the graph's betweenness

[6] `https://graph-tool.skewed.de/`

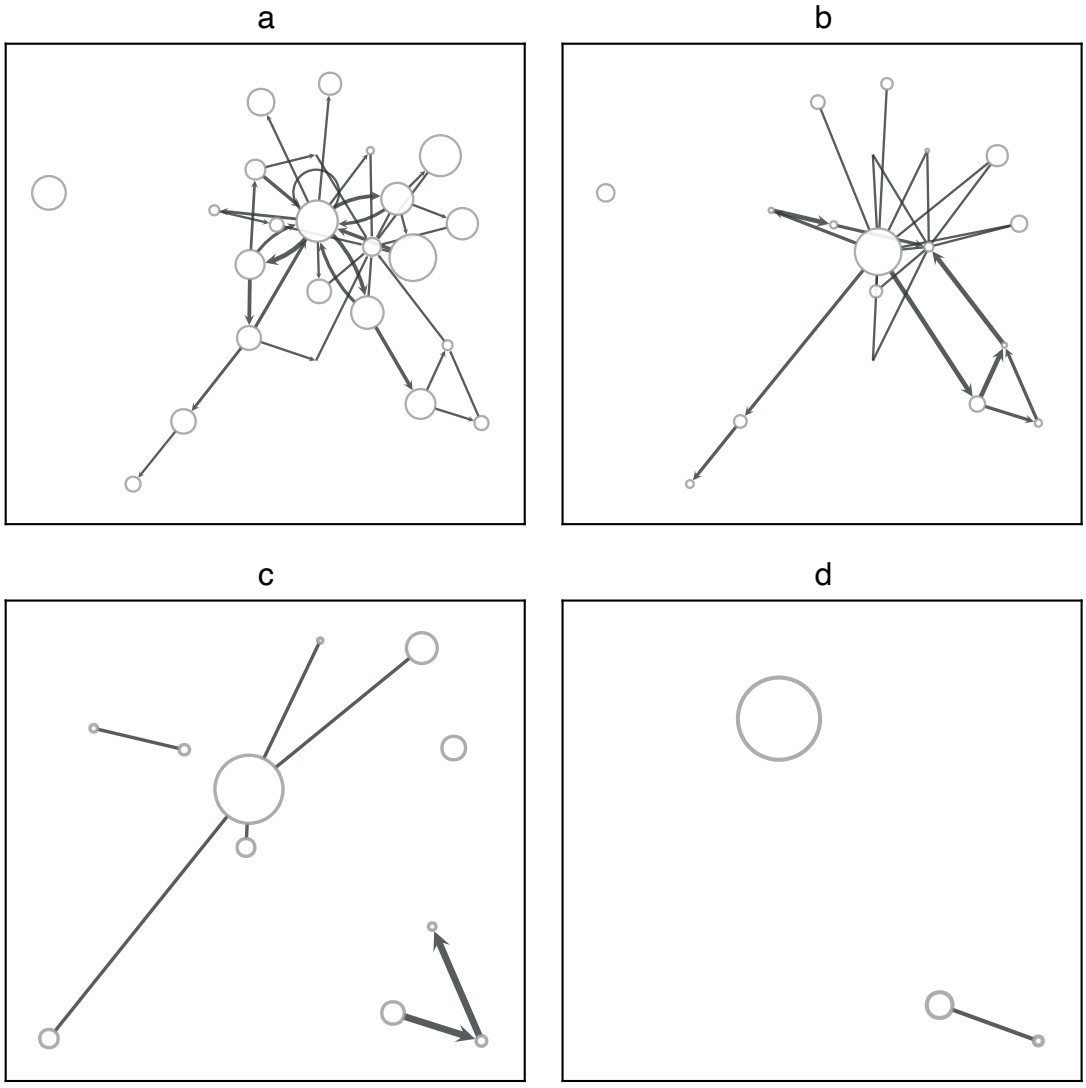



Figure 3.6: **Graph reduction**. **(a)** shows the original graph; **(b)** condenses strongly connected components and self-loops as per [52]; **(c)** condenses as in (b) and also prunes edges which are not part of the minimum spanning tree; **(d)** applies (c) twice. Each view preserves the relative positioning from the original graph, scaled to fit the new graph.

centrality. Repeated application of graph condensation and filtering by the minimum spanning tree provides an extreme reduction in the graph's complexity.

We select an arbitrary set of graph properties as a feature vector (**average edge betweenness**, **average vertex size**, **vertex degree histogram**, and **vertex degree** normalized by **vertex size**) and then train a random forest classifier to predict the class of the function based on this vector. For each of these properties, our feature vector consists of its **mean** and its **standard deviation**, for a total of 8 features per graph.

### 3.4 Discussion

Unlike related work in this area which may involve obfuscated source code, our approach focuses specifically on the identifying library functions. While these functions are often small, they are used frequently in application code by both benign and malicious developers. Identifying library functions quickly and accurately aids in the overall task of binary analysis, because once those functions are identified, analysis can proceed to the more interesting aspects of a particular section of code, such as finding malware or stolen intellectual property.

In the case where binary code is dynamically linked with system libraries, our method has little to offer, because those libraries can be analyzed just one time even if there are many dynamically linked applications that use them. However, if desired our approach could be used to analyze an existing library. If such analysis reveals an unknown function, this could imply a library function missing from our model's dataset, or possibly a library function which is implemented in an unusual manner. It is reasonable to audit shared library code closely, because a vulnerability in a shared resource impacts many other applications.

Unlike related work which uses data augmentation techniques inspired by computer vision [7], [11], our method attempts to preserve the semantics of the source material. Automated obfuscation using a tool like Tigress [17], [50] is repeatable and testable, permitting verification of program semantics both before and after transformation. However, in this work we do not run any behavioral tests of the code either before or after obfuscation, instead we rely on the less strict measure of

compilation success. We only include in our method those transformations that result in source code that can compile without any errors.

One hypothesis which we examine in this work is whether graph metadata affects the outcome of a machine learning model. Compared to PV-DM models, graph-based models convey relationships between parts explicitly and succinctly. Therefore, we expect that the addition of graph attributes to a model should improve its accuracy at the cost of increased complexity during data preprocessing, because obtaining these graph characteristics requires more domain knowledge of the assembly code.

### 3.5 Results

Table 3.3: Individual model accuracy on test set

| model | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | **mean** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| $\frac{correct}{total}$ | 0.68 | 0.68 | 0.65 | 0.69 | 0.69 | 0.68 | 0.7 | 0.69 | 0.68 | 0.69 | **0.68** |

Table 3.3 summarizes the accuracy of each of the individual classifiers. To measure accuracy, we take a simple count of all correct predictions, and divide by the total number of predictions. Each of the 10 models is 68% accurate on average. But because each of the 10 sub-models trained on a *different* 90% of the original corpus, voters strong in one area can compensate for other voters weak in that area. During experimental evaluation, this enabled the model to achieve 100% accuracy on a set of test samples which were withheld from the training set.

While high accuracy on a training sample is encouraging, a more realistic measure of voting classifier accuracy is predicting the correct class for a new function implementation which was never seen in either training or testing. To test this, we generate a small dataset with a different set of Tigress transformations (`Split` then `Flatten`) and compilation options (`-O2`). While the functions themselves are still from `musl`, these compiled binaries are new to our models.

Table 3.4 shows the results for these new function implementations. The simple accuracy of the voting ensemble for unseen data is **74%**, and Cohen's kappa coefficient [53] is **0.73**. In this table,

Table 3.4: Classifier results: new data

| actual | voting prediction (74%) | graph prediction (64%) | graph override (74%) |
|---|---|---|---|
| abs | **abs** | **abs** | **abs** |
| acos | inet_addr | abs | abs |
| asin | acos | **asin** | **asin** |
| atan2 | **atan2** | strtok | **atan2** |
| ceil | floor | **ceil** | floor |
| cos | sin | asin | asin |
| daemon | **daemon** | abs | **daemon** |
| exp | sin | tan | sin |
| floor | **floor** | **floor** | **floor** |
| fread | **fread** | **fread** | **fread** |
| inet_addr | **inet_addr** | **inet_addr** | **inet_addr** |
| inet_aton | **inet_aton** | **inet_aton** | **inet_aton** |
| isalnum | **isalnum** | **isalnum** | **isalnum** |
| kill | **kill** | **kill** | **kill** |
| memccpy | **memccpy** | strstr | **memccpy** |
| memcmp | **memcmp** | **memcmp** | **memcmp** |
| memmem | **memmem** | **memmem** | **memmem** |
| readdir | **readdir** | **readdir** | **readdir** |
| signal | **signal** | **signal** | **signal** |
| sin | **sin** | asin | asin |
| stpcpy | strncpy | inet_aton | strncpy |
| stpncpy | **stpncpy** | **stpncpy** | **stpncpy** |
| strchr | **strchr** | **strchr** | **strchr** |
| strcpy | strncpy | **strcpy** | strncpy |
| strncpy | **strncpy** | strchr | **strncpy** |
| strstr | **strstr** | **strstr** | **strstr** |
| strtok | **strtok** | **strtok** | **strtok** |
| tan | **tan** | memcmp | **tan** |
| vfprintf | acos | asin | acos |
| wait4 | **wait4** | **wait4** | **wait4** |
| waitpid | **waitpid** | **waitpid** | **waitpid** |

correct predictions by the classifiers are in **bold face**. Figure 3.7 shows the predicted versus actual label for the test set (10% of total) using a Random Forest classifier on the graph metadata. The overall accuracy of this classifier using the original graph structure is **64%**. However, we still see **64%** accuracy with this model when analyzing the **reduced** graph data. Reducing the graph structure using the methods of Figure 3.6 has **no impact** on accuracy, but uses less memory. As datasets grow larger, this reduction technique may become essential.

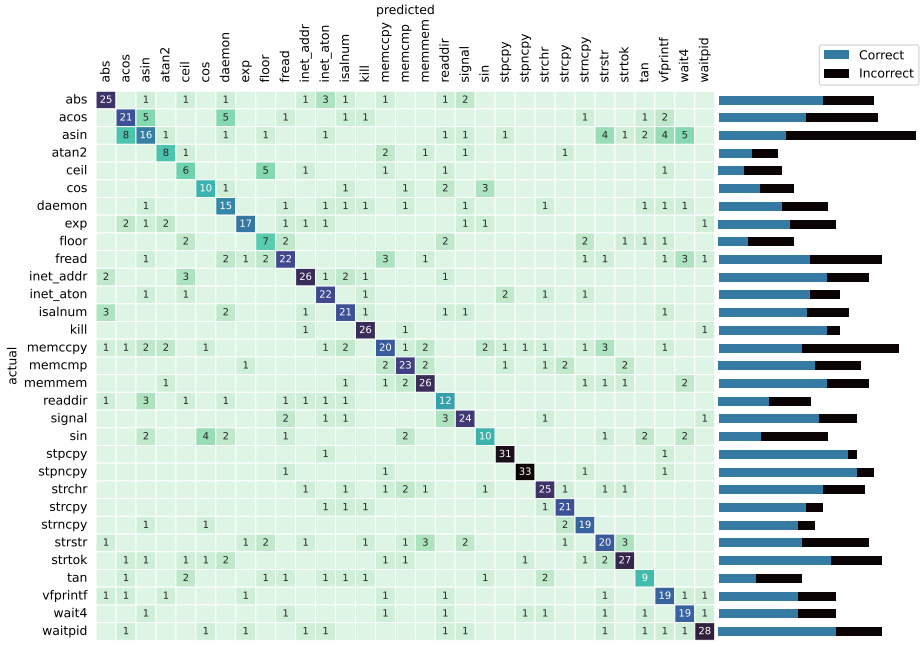



Figure 3.7: Confusion matrix for graph metadata classifier (validation data)

The graph metadata classifier accuracy is still acceptable at **67%** when presented with unseen test data. Figure 3.8 shows predicted versus actual labels for this small test dataset. We combine these two classifiers by **overriding** the voted prediction with the graph prediction when the voting classifier was "uncertain", based on a simple heuristic (more than 4 different votes and the winner received less than 5 votes). With this simple heuristic, the combined voting/graph classifier remains **74%** accurate. Contrary to our expectations, the addition of this graph metadata did not improve the overall quality of result.

Compared to asm2vec, our work succeeds at classifying highly obfuscated code, as illustrated in Table 3.5.

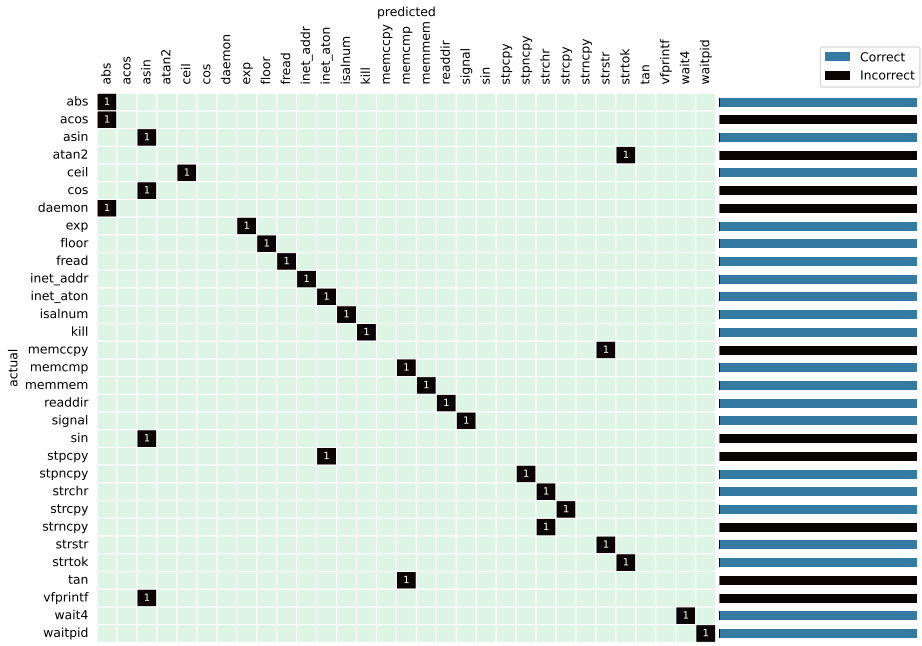



Figure 3.8: Confusion matrix for graph metadata classifier (test data)

Table 3.5: Comparison with Asm2Vec

| | EncodeLiterals | Virtualization | JIT | 3 transformations |
|---|---|---|---|---|
| Asm2Vec | **92.7%** | 35% | 45% | 0% |
| ours | 74% | **74%** | untested | **74%** |

### 3.6 Conclusions and Future Work

We demonstrate source code augmentation through obfuscation suitable for training different types of machine learning models. Our voting classifier model classifies new variations of given functions with 74% accuracy, whereas a graph-metadata based classifier classifies this same test data with 64% accuracy. Combining the predictions of both classifiers with a simple voting heuristic neither improves nor degrades this 74% accuracy. This approach tolerates obfuscated source code and stripped binary files, unlike asm2vec, whose authors note that "after applying three obfuscation techniques at the same time, asm2vec can no longer recover any clone". Our approach can not only discriminate between similar versus dissimilar samples, but also classifies multiple functions with accuracy exceeding that of asm2vec when the functions were subject to obfuscation.

```
1       endbr64
2       push r13
3       mov eax, CONST
4       push r12
5       push rbp
6       lea rbp, [rip]
7       push rbx
8       sub rsp, CONST
9       movsd qword [rsp + CONST], xmm0
10      lea r13, [rsp + CONST]
11      lea r12, [rsp + CONST]
12  LABEL12:
13      cmp rax, CONST
14      cjmp LABEL12
15      movsxd rdx, dword [rbp + rax*CONST]
16      add rdx, rbp
17      jmp rdx
18      ret
```

acos (actual)

```
1       endbr64
2       mov qword [rsp + CONST], rdi
3       lea rdx, [rsp + CONST]
4       lea rsi, [rsp + CONST]
5       lea rdi, [rsp + CONST]
6       call LABEL6
7   LABEL6:
8       move eax, dword [rsp + CONST]
9       test eax, eax
10      cjmp LABEL9
11      move eax, dword [rsp + CONST]
12      add rsp, CONST
13      ret
14  LABEL9:
15      mov eax, CONST
16      add rsp, CONST
```

inet_addr (predicted)

Figure 3.9: Mistaken function example

Our approach has some limitations, however, and there is room for improvement. Not all Tigress transformations were successful in producing output, a problem also encountered by the asm2vec authors. We were able to work around some of these errors by carefully crafting the commands passed to the compiler, but some such errors remain. These are due to the `musl` library's usage of newer C syntax features not yet supported by Tigress' C parsing engine. In these

cases, a particular function was not transformed by a particular Tigress transformation, and so that function-transformation pair was not included in our dataset. Manual fixes limit the generality of this approach because they require familiarity with the build system. This is not a problem for in-house analysis, but it could be a challenge for analyzing unknown code.

We also note that there are some examples in the unseen data which our voting classifier never guessed correctly (e.g. `vfprintf`, `acos`, and `strcpy`). With the `strcpy` sample, the classifier predicted `strncpy(6) stpcpy(3) stpncpy`. We believe these predictions are good approximate matches to the `strcpy` function, and would therefore be valuable to a human using our approach as a reverse engineering aid. The predictions for `vfprintf` (`asin(5)` and `acos(5)`) are not close at all, and would be misleading to a human. This instance of `vfprintf` happens to contain no labels or jumps, which could indicate a limitation of our data augmentation strategy. The `acos` example was mistaken for `inet_addr`, so we compare them side-by-side in Figure 3.9. These two samples do not have any major similarities other than length, so more analysis is needed to determine how the model can better handle situations like these. Figure 3.10 shows the distribution of basic blocks in our data before and after the graph condensation and pruning. This distribution is acceptable for analysis, because such skew is likely to occur in real-world code as well as synthesized code, even though library functions in particular tend to have fewer nodes and edges.

In future work we plan to evaluate the effectiveness of graph coarsening in contrast to the simpler graph reduction we applied in this work. Coarsening methods such as these preserve spectral properties of the original graphs, unlike our minimum spanning tree and condensation approach, and may better preserve the structure of small graphs. We also plan to incorporate graph embedding and recent graph neural network techniques to increase the effectiveness of our classifiers. Finally, while the workflow we developed so far is useful for detecting specific functions within an obfuscated binary, we plan to further analyze whether it can distinguish between functions it has never seen in **any** form versus functions it has seen in a modified form.

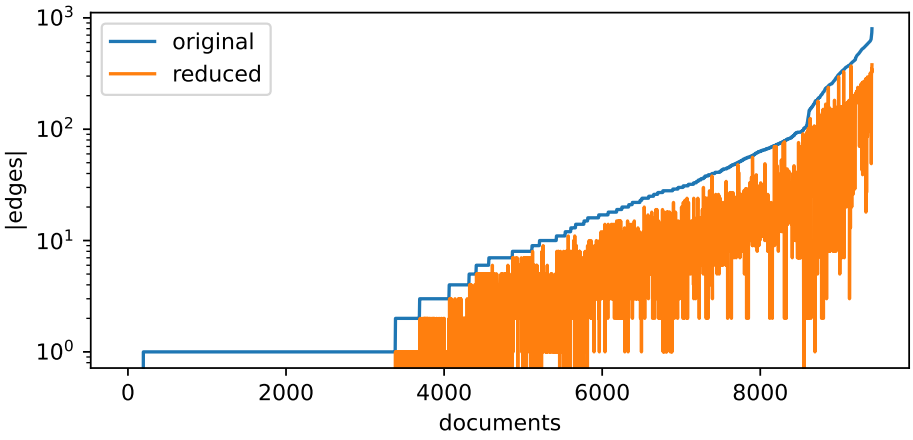



Figure 3.10: Jump-Label distribution

# CHAPTER 4

# FUNCTION CLASSIFICATION FOR OBFUSCATED BINARY CODE

Modern software builds on a foundation of opaque binary code, which makes analysis difficult. Inside this code there may be vulnerabilities or stolen intellectual property, so effective analysis tools are an essential part of forensic software analysis and reverse engineering. With sufficient quantities of the right kind of data, machine learning models could help find these vulnerabilities, but obtaining sufficient training data remains a stubborn challenge. In this chapter, we propose a novel approach to data augmentation which effectively addresses the data shortage problem. We augment data using obfuscation for source code, diverse compilation for binary code, and by extracting basic blocks and sampling random walks through their resulting control flow graphs. These forms of augmentation not only make our approach robust to obfuscation but also enable us to use simpler classifiers. We evaluate multiple models which use different representations of the same data to compare and contrast what forms of data augmentation work best in different scenarios. Finally, this chapter concludes with an application for real-world program analysis by using a model to detect known functions within utility programs as well as malware This application demonstrates how our work can improve productivity for the reverse engineering practitioner.

## 4.1 Introduction

The problem of determining similarities between two different programs is, in general, unsolvable. Even answering the question "does this program halt?" is beyond the reach of current theory. However, by relaxing the question ever so slightly, we arrive at an approach that is not only tractable, but useful. We train machine learning models to recognize a known set of functions, then evaluate unknown code based on its similarity to one of the known functions. This reduces the scope of our problem from the unsolvable "what is the meaning of unknown code X?" to the tractable "how similar is unknown code X to known code Y?" Answering this question with machine learning

requires all of: (a) sufficient data, (b) a model to train, and (c) some metric by which to judge binary similarity; this work addresses all of these points.

In recent decades, machine learning has undergone a revolution, with new techniques driving advances in natural language processing (NLP), computer vision (CV) and other domains. Many of these advances rely on massive datasets for training. However, when it comes to analyzing compiled code, lack of data remains a serious problem. While there have been attempts [54], [55] to aggregate malware samples and compiled binary program data, suitable sources for the most part remain fragmented.

In this work, we apply machine learning to detect known functions within obfuscated and stripped binary code. This is relevant for tasks ranging from malware detection to intellectual property protection. Despite the scarcity of real-world training data and the unique properties of compiled binary data, we show that it is possible to train high quality models on consumer-grade hardware quickly. In contrast to approaches such as [38], [56] is that rather than detecting known patterns of code, we are interested in detection of code that is similar but *not* identical to known code.

To obtain sufficiently large quantities of training data, we propose a method of data augmentation based on source code obfuscation and diverse compilation. Through additional domain-specific feature engineering, we generate a large number of example data while preserving the unique structural characteristics of compiled code.

We provide different models each requiring varying degrees of data preprocessing, and evaluate them all on the same data to contrast their strengths and weaknesses.

For determining function similarity, modern reverse engineering tools are typically limited by their reliance on heuristics [57], assumptions about instruction set architecture, or avoidance of obfuscated code. We focus on what we consider to be the most difficult case: intensely obfuscated binary code that has been stripped of all symbols. In this work we focus on the x86-64 instruction set architecture (ISA), but the same technique applies to any architecture for which a disassembler exists. The similarity metric differs slightly for each of our models, but all are based on the same premise: train on a diverse set of representative samples to learn to classify the implicit rather than

explicit patterns in the data.

The contributions of this paper are as follows. First, we describe obfuscation-based data augmentation which retains essential syntactic and semantic features of compiled code. Second, we measure the utility of adding domain-specific engineered features to our baseline approach. Finally, we conduct a case study applying these techniques to locate known functions within stripped real-world binary programs.

### 4.1.1 Related work

Traditional approaches to binary analysis relied heavily on feature engineering, applied domain knowledge, and carefully crafted heuristics. Detecting known sequences of bytes within a binary is sufficient insight for certain applications. When the content of the functions of interest is static, fixed pattern matching as in [56] can help detect known functions. The reverse engineering software IDA Pro has a plugin called FLIRT [38] that builds on this approach. FLIRT uses a database of known function "fingerprints" and searches for matching fingerprints within the analyzed code. Unfortunately, this fingerprinting approach can be spoofed by an adversary [39]. By assuming the binary is being analyzed in IDA, the attacker constructs fake fingerprints that are identified by FLIRT as benign. However, these fake fingerprints hide malicious code. IDA is a market leader in reverse engineering software, so IDA-specific attacks are worthwhile to an attacker, however fingerprint-based matching from any vendor is inherently vulnerable to this type of attack.

More recent deep learning and embedding learning methods for binary analysis still rely on assumptions about the input data that are not universally true. Asm2Vec [13] achieves good results for plain code but relies on the presence of debug symbols in the code. The accuracy of Asm2Vec is highest in the case of analyzing non-obfuscated code, but deteriorates when applied to code with increased amounts of obfuscation. Gemini [58] tolerates differences between instruction set architectures, but depends on control flow graph (CFG) similarity, making it less useful in the presence of obfuscation techniques which alter the program's CFG. Nero [59] uses augmented call sites to predict names of the called procedures, and [60] uses a method of encoding the abstract syntax

tree in a way that can be learned by a long short-term memory (LSTM) network. Massarelli et al. employ a token embedding model using the word2vec [14] methods and suggest the use of self-attention [61] as in [62] improves over the results of Gemini. They test on non-obfuscated binary data only. Likewise, [63] focuses on learning an assembly language model based on instruction embeddings. All these approaches start by building vector embeddings of their input. Embeddings are good for analyzing obfuscated code because minor variations between similar examples may end up nearby in the embedding space. However, it is important to note that this robustness is highly dependent on the diversity of training data.

Graph machine learning methods help predict relationships among entities in irregularly-shaped data such as social networks, road networks, or molecular structures [64], but are difficult to run efficiently. Marcelli et al. suggest that graph-based neural networks (GNNs) are better at function similarity than simpler fuzzy hashing approaches [54], and suggest combining GNN-based approaches with instruction-level encoders. Some embedding methods, such as [13], [23], [58] rely on extracting CFG information from the binary, typically by relying on reverse engineering software such as IDA Pro [65].

In their survey, Haq et al. [55] identify different binary code similarity approaches. They find that most approaches benefit by using vector representations, and a majority of papers focus on one-to-many comparisons. A characteristic shared by several approaches in this survey is that the datasets used tend to either focus on cross-language/cross-architecture similarity (as in [60]), or malware/benign binary classification problems.

The work of Jin et al. is closely aligned to our problem [66]. Their SymLM model aims to predict the name of a function given stripped binary input data. SymLM improves on previous work by David et al. [59], which predicts symbol names from stripped binaries. The improvement of SymLM is due to the inclusion of execution-aware metadata, which results in a more complex model architecture. They also train on a large number of functions across 16,027 different binaries, but with only four obfuscation options.

Pei et al. build on SymLM with their work, SymC [67]. They demonstrate improvements over

SymLM using a model that preserves symmetries based on a group theoretic representation of instructions. SymC tests the same four obfuscation options as SymLM.

A binary analysis task related to obfuscation is the control flow transformation work by Yakdan et al. [68]. This could be one step serving a larger de-obfuscation task.

Yu et al. [69] propose a model that combines embeddings in the instruction, block, and graph level. They represent the CFG as an adjacency matrix add an Convolutional Neural Network (CNN) alongside BERT [70] which considers token and block embeddings. By merging the outputs of these two models, they are able to consider features at different hierarchical layers. Their evaluation task is binary similarity at two different optimization levels (`-O2` and `-O3`) and across two architectures (x86-64 and ARM), and does not include any tasks related to obfuscation. Another work which utilizes an embedding followed by a deep neural network is described in [71, Chap. 4]. This work achieves around 86% accuracy in distinguishing 12 function classes in a test set of 1000 samples, after training on 10800 samples.

### 4.1.2 Research questions

*What makes a model robust to obfuscation?*

The landscape of related works is fragmented by a diversity of approaches and a diversity of different datasets. Some focus solely on malware, others on plagiarism detections, and still others on cross-architecture similarity. While some datasets include a small amount of slightly obfuscated code, ours focuses on obfuscation as the primary mode of data augmentation. We also use more severely obfuscated code by composing multiple obfuscation types in a single datum.

*Does training on obfuscated code improve predictions on non-obfuscated code?*

When considering the predictive strength of a model $M(\mathrm{train}, \mathrm{test})$ with a full set of obfuscated code $o$, a subset of the obfuscated code $s$ or plain code $p$, so we consider the combinations in Table 4.1

We consider scenario 3 the most challenging from an analysis perspective but also representative

Table 4.1: Train and test combinations

| Train Dataset | Test Dataset |
|---|---|
| Obfuscated | Obfuscated |
| Obfuscated | Plain |
| Plain | Obfuscated |
| Plain | Plain |
| Subset+Plain | Obfuscated |
| Subset+Plain | Plain |

of an important threat model. In this threat model, a dedicated attacker has obfuscated their code, but the under-prepared defender only has the plain code for comparison. Likewise scenario 2 corresponds to a well-prepared defender with a corpus of obfuscated training data versus a lax attacker using no obfuscation at all. Scenarios 1 and 4 indirectly illustrate the effects of adding obfuscation to the dataset. In our experiments, we found that obfuscation tools do not necessarily work with every possible function. Scenarios 5 and 6 are most relevant for this kind of situation because they represent a dataset which is only partially augmented with obfuscated training examples.

*How does graph representation impact model performance?*

Finally, we use assembly language's basic block graph structure to synthesize a representation that reflects the binary program's control flow to add some of the runtime semantics to our static analysis approach. We evaluate the same models on this representation to determine how this graph structure affects model performance.

## 4.2 Methods

### 4.2.1 Dataset

A key challenge when applying deep learning models to any problem is obtaining sufficient quantity of high-quality data. For compiled binary programs, there are typically only a handful of different versions to choose from. These different versions generally only exist in order to support the most common three or four operating systems, and such a small number of versions is insuffi-

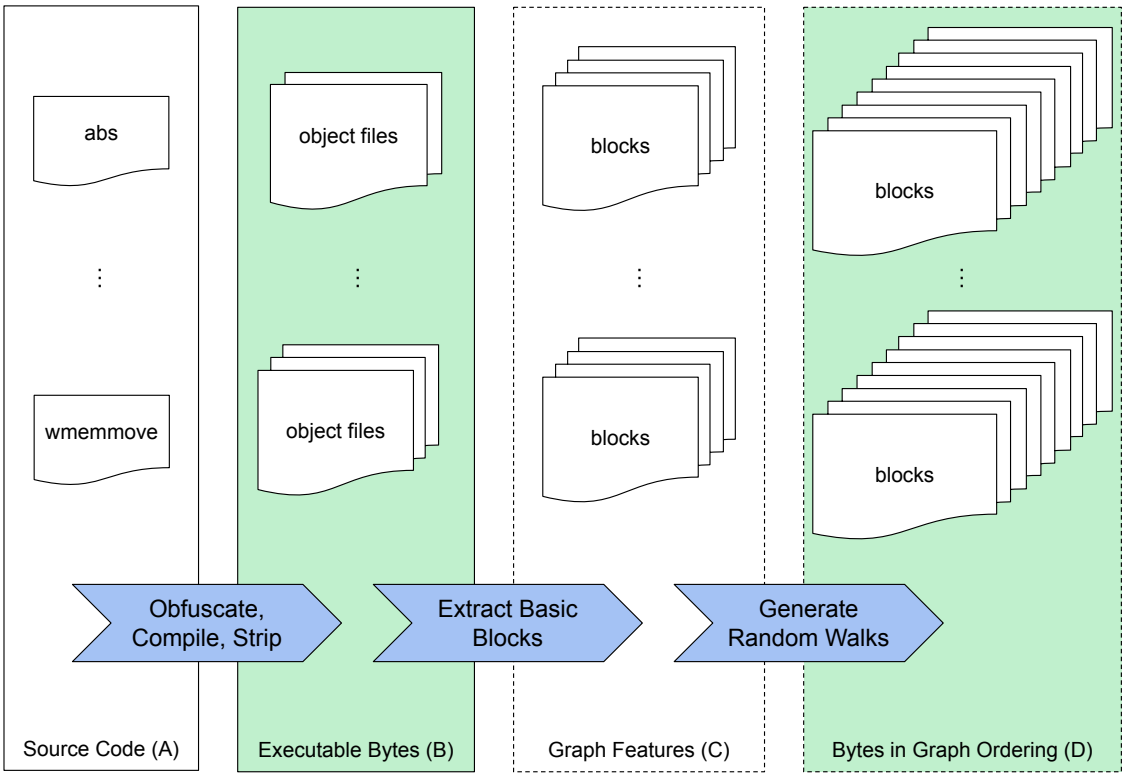



Figure 4.1: Dataset generation stages

cient for training a deep learning model. We address this shortage of binary data through the use of obfuscation, diverse compilation, and feature engineering to augment existing data [72].

Figure 4.1 summarizes the construction of our dataset. Our approach begins with the source code from a set of *functions of interest* corresponding to Figure 4.1 (A). We use 25 functions from the open source musl[1] C library implementation (version 1.2.4) as our running example. Some functions are conceptually related, such as the pair (`floor`, `ceil`), the assorted trigonometric functions (`acos`, `atan2`, `sin`, `tan`, ...), and the string related functions (`strchr`, `strcpy`, `strtok`, ...). Others such as `daemon` and `inet_aton` are not obviously similar to any of the others. Figure 4.1 only shows two example functions for clarity.

To transform these sources into Figure 4.1 (B), we first obfuscate the source code of these functions using a number of the following Tigress[2] transformations:

- **Plain**: no obfuscation

[1]The musl C library: `https://musl.libc.org/`
[2]Tigress C Obfuscator: `https://tigress.wtf/`

- **Flatten**: replace control flow with a jump table or switch statement
- **Split**: break up a function into multiple parts
- **EncodeArithmetic**: make existing arithmetic expressions more complex
- **RndArgs**: randomize the order of function arguments
- **Inline**: inline the function body at a function call site
- **Merge**: combine multiple functions

For this set of transformations $T$, we take up to four obfuscating transformations as in $\binom{|T|}{i}_{i=1}^{4}$, resulting in a new sequence of transformations such as (Split), (Split, Flatten, Split), and (Merge, Inline, RndArgs, RndArgs, Merge). Because there are more combinations of $\binom{|T|}{4}$ than $\binom{|T|}{1}$, our dataset skews toward highly-obfuscated code more than lightly-obfuscated code. This preference for highly obfuscated code contrasts with related works like [13], [66], [67], [73] which only use small amounts of obfuscation. These works are notable within the broader landscape of binary similarity literature for using any amount of obfuscation, but compared to our approach we consider a single obfuscating transformation to be "lightly" obfuscated. While our dataset is primarily highly obfuscated code, we also include both lightly obfuscated and plain examples.

Each obfuscated or plain source file is then compiled using one of the following groups of options:

- **Plain**: no change to musl default compilation options
- **Loops**: -floop-parallelize-all -ftree-loop-if-convert -funroll-all-loops -fsplit-loops -funswitch-loops
- **Codegen**: -fstack-reuse=all -ftrapv -fpcc-struct-return -fcommon -fpic -fpie
- **Safety**: -fsanitize=address -fsanitize=pointer-compare -fsanitize=undefined -fsanitize-address-use-after-scope
- **Os**: -Os (optimize for size)

- **O3**: -O3 (optimize for speed)

Next we use the binary analysis tool angr [29], [74], [75] to extract a control flow graph (CFG) from the basic blocks of every object file for every function (Figure 4.1 (C)), then sample random walks of these CFGs (Figure 4.1 (D)). This stage lets us model a linear, static sequence of instructions as a set of pseudo-execution trace subgraphs, and allows our dataset to capture potential runtime control flows without incurring the computational cost or security risk of running potentially malicious code.

Table 4.2 illustrates the relationship between these subgraphs and the extracted control flow graph. Each row shows a list of instructions that make up a basic block (truncated with ... for brevity). This example comes from the `ceil` function with the Tigress transformations (Split, Split, Branch) and compiler optimization group **O3**. The random walk shown here traverses the blocks numbered **1**, **2**, **4**, **6**, and **8** in sequence. At block **1**, there is a conditional jump to **0x400119**, but in this random walk, that jump is not taken. Control then falls through by default to block **2**, whose conditional jump to **0x400128** is taken. This particular random walk has length 5 and therefore stops at block **8**, but other walks traversing block **8** could take the conditional jump back to block **1** and demonstrate looping control flow.

$$|C_i| = \min(|C_i|, \mu + \sigma) \tag{4.1}$$

After removing transformed sources that do not compile and over-optimized binaries, we balance the training data according to Equation 4.1. This ensures no class will have more examples than one standard deviation above the mean number of examples for all classes, reducing extreme degrees of class imbalance.

When it comes to dataset preparation for binary similarity, most related works employ diverse compilation. This usually entails using different compilation options, multiple compilers, or different instruction set architectures in the case of cross-architecture similarity analysis. However, Table 4.3 highlights the key problem of using only compiler options to add diversity to binary

Table 4.2: Addresses and partial listing of instructions for disassembled object file. Block number and addresses **bold** for the random walk (1,2,4,6,8). The last column indicates whether the random walk takes the conditional jump.

| block | address | instructions | jump? |
|---|---|---|---|
| 0 | 0x400000 | endbr64 0x400004 … 0x400012 ret | |
| **1** | 0x4000fd | cmp dword ptr … jg **0x400119** | no |
| **2** | 0x40010d | pxor xmm2, … jp **0x400128** | yes |
| 3 | **0x400119** | add rsp, 0x… r13 0x400122 ret | |
| **4** | **0x400128** | lea r13, [rsp … call **0x40013b** | yes |
| 5 | 0x400117 | jne 0x40012 0x400128 | |
| **6** | **0x40013b** | cmp dword ptr … jle **0x400170** | no |
| 7 | **0x400170** | mov rdi, r13 … call 0x400178 | |
| **8** | 0x400145 | movsd xmm1, … jbe **0x400119** | no |
| 9 | 0x400178 | lea rsi, [rsp … call 0x400185 | |
| 10 | 0x40015f | addsd xmm0, … jmp 0x400119 | |
| | | … | |
| 27 | 0x400096 | nop word ptr … cs:[rax + rax] | |
| 28 | 0x4000af | nop | |
| 29 | 0x4000cf | nop | |
| 30 | 0x4000d0 | endbr64 0x4000d4 … call 0x4000fd | |

data: some functions are so small they are frequently optimized into nothing. For example, only two non-obfuscated versions of `abs` survived our requirement that each compiled example must contain the function of interest before being stripped of symbols. Conversely, when generating obfuscated variations of functions our only limits are disk space and patience.

Figure 4.2 shows that the majority of generated object files are small, but the obfuscated object files have a long tail of outliers with large file sizes. This figure also contrasts the small quantity of **Plain** compared to the much larger **Obfuscated** data.

Some of our models use a bag-of-words approach where byte values are treated as "words". We count the bytes of each example to produce a vector $\mathbf{V}^{256}$ where each index $\mathbf{V}_i$ denotes the number of times byte $i$ appears in document $D$. This experiment reads the object file bytes and creates a training example by counting the occurrence of each byte value. Figure 4.3 visualizes a portion of this data. Each row is one vector $\mathbf{V}^{256}$ of byte counts, with functions grouped alphabetically. Similarities between different samples for the same function appear as similarly-colored horizontal bands. The normalized plot normalizes by dividing the column sum, and to increase visual contrast

Table 4.3: Total numbers of files and their basic blocks for each function. Subscripts $o$ and $p$ indicate **Obfuscated** and **Plain** data, respectively.

| function | $\text{File}_\text{o}$ | $\text{File}_\text{p}$ | $\text{Block}_\text{o}$ | $\text{Block}_\text{p}$ |
|---|---|---|---|---|
| abs | 843 | 2 | 15284 | 19 |
| acos | 1168 | 6 | 111426 | 151 |
| asin | 1168 | 6 | 115976 | 150 |
| atan2 | 1084 | 6 | 252536 | 428 |
| ceil | 1168 | 6 | 141695 | 134 |
| cos | 1168 | 6 | 337723 | 187 |
| daemon | 1030 | 6 | 178383 | 241 |
| floor | 1168 | 6 | 141259 | 133 |
| inet_addr | 1140 | 5 | 61834 | 49 |
| inet_aton | 1140 | 6 | 62656 | 272 |
| isalnum | 1222 | 6 | 26814 | 70 |
| memccpy | 1152 | 6 | 156176 | 176 |
| memcmp | 1072 | 4 | 97352 | 63 |
| memmem | 1222 | 6 | 312598 | 775 |
| sin | 1168 | 6 | 430119 | 208 |
| stpcpy | 1156 | 6 | 38219 | 141 |
| stpncpy | 1164 | 6 | 48265 | 191 |
| strchr | 1129 | 5 | 31175 | 42 |
| strcpy | 1032 | 4 | 16833 | 22 |
| strncpy | 1074 | 5 | 17212 | 28 |
| strstr | 1222 | 6 | 366570 | 857 |
| strtok | 1168 | 6 | 40486 | 98 |
| tan | 1168 | 6 | 249496 | 129 |
| utime | 1153 | 6 | 26297 | 67 |
| wmemmove | 1128 | 5 | 57067 | 97 |

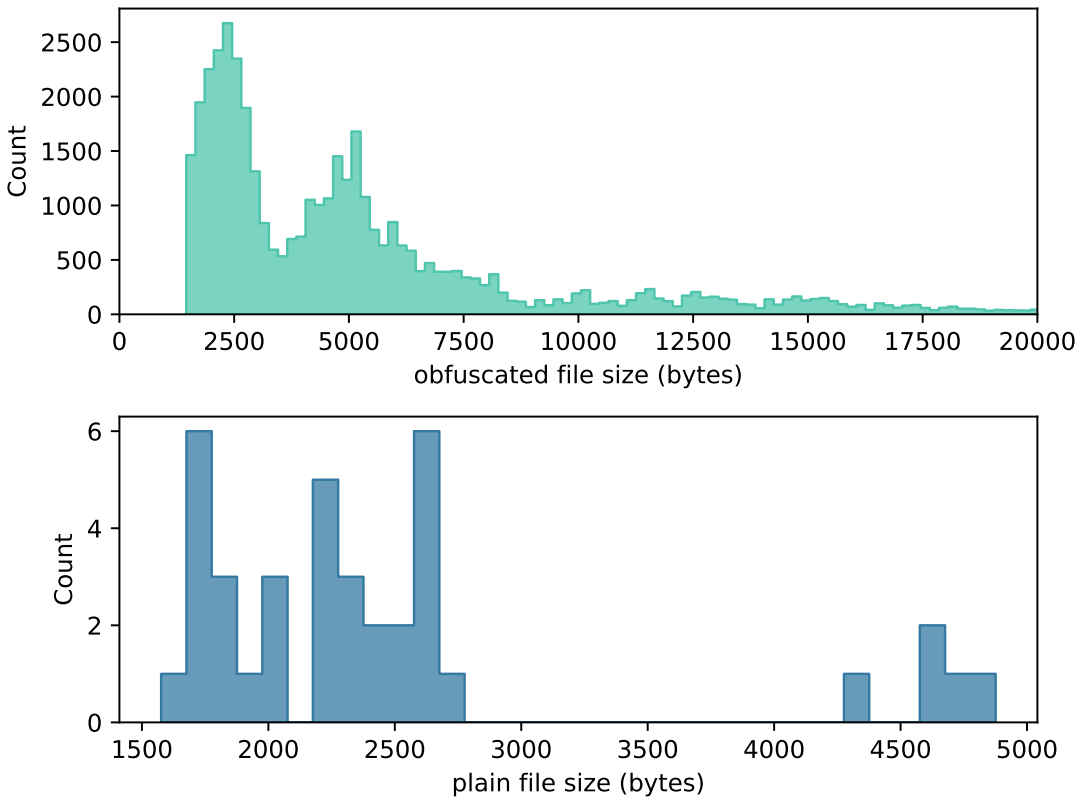



Figure 4.2: Distribution of object file sizes

the max normalized/max plot additionally divides this value by the max count per row. This figure makes clear that while different functions are generally visually distinct, there can also be large differences between instances of the *same* function, which will confound any model using this bag-of-bytes representation of the data.

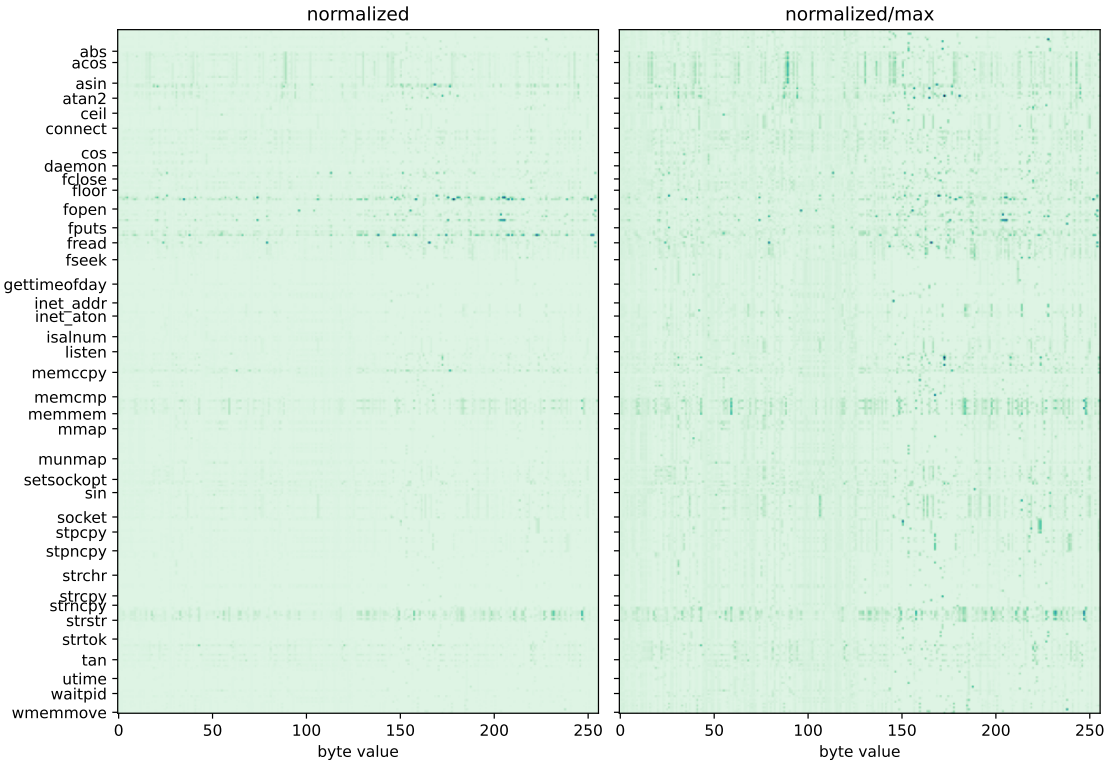



Figure 4.3: Byte counts

We use angr to extract basic blocks from the binary files. To get textual representation of this binary data, we use angr's version of the Capstone[3]. Next we normalize punctuation so `mov rbx, [rax+0x8]` becomes `mov rbx , [ rax + 0x8 ]`. Finally we normalize hex constants based on their length. For example, `0xf` becomes `0x1` (because `f` is one character), `-0x1a23` becomes `-0x4` (`1a23` is 4 characters), and `0x1234567812345678` becomes `0x10` (`1234567812345678` is 16 characters long, and 16 is `10` in hex). This reduces the domain of numeric constants tokens by $\log_{16}$ and in practice acts as dimensionality reduction because most disassembled hex literal values fall within a small range of sizes. These disassembled textual

[3]Capstone disassembler: `https://www.capstone-engine.org/`

tokens become another representation of the data which we use to compare models.

Finally, we categorize this dataset into three logical sections. The majority of samples are **Obfuscated**. We also include a small number of **Plain** samples with no obfuscation applied. Finally, we extract a **Subset** of the obfuscated data by selecting examples which belong to only half of the function classes. These three sections will be used in training and validation later.

### 4.2.2 Models

The following sections describe the models we use to classify functions within obfuscated binary code. We developed multiple models to compare for this work: k-Nearest Neighbors (kNN), Random Forest, FastText (an NLP text classifier), and a custom embedding-based model implemented in PyTorch.

*k-Nearest Neighbors*

As a baseline, we employ a deflate[4]-based approach inspired by [76], which uses *normalized compression distance* (NCD, given in Equation 4.2) as a similarity measure between a pair of documents $D_1$ and $D_2$. There is only one hyperparameter for this model: $k \in \{1, 3, 5, 9\}$. The model accepts binary object files as input, and chooses the $k$ nearest neighbors to a given sample according to their NCD to determine an object file's label.

$$\mathrm{NCD}(s_1, s_2, s_{12}) = \frac{s_{12} - \min s_1, s_2}{\max s_1, s_2} \tag{4.2}$$

Where $s_1$ and $s_2$ are the sizes of compressed documents $\mathrm{deflate}(D_1)$ and $\mathrm{deflate}(D_2)$, and $s_{12}$ is the size of the compressed concatenation $\mathrm{deflate}(\mathrm{concat}(D_1, D_2))$.

*Random Forest*

We train a second classifier on raw object file bytes, but this time using a Random Forest ensemble model. The hyperparameters we used in this experiment include:

[4]Deflate compression algorithm: `https://zlib.net/feldspar.html`

- **n_estimators** $\in \{40, 80, 100, 120\}$
- **n_jobs** = -1 (use all 18 processors in the development environment)
- **class_weight** = "balanced"
- other hyperparameters use scikit-learn default values[5]

*Random Forest$_{tokens}$*

This classifier uses the same hyperparameters as the Random Forest model, but instead of using raw bytes as input, we first disassemble the tokens and then construct a bag of words for the text tokens. We also use this text token representation for the FastText model.

*FastText*

The disassembled tokens should embody more structure than the raw byte counts. If our models can take advantage of this structure they should have better performance. We use **FastText** [77] version 0.9.2 to learn word representations of the assembly language tokens. This model uses a continuous skipgram approach to maximize the log-likelihood given by Equation 4.3

$$\sum_{t=1}^{T} \sum_{c \in C_t} \log(w_c | w_t), \tag{4.3}$$

where $C_t$ are indices of words around word $w_t$. Our implementation uses the following non-default hyperparameters:

- **threshold** $\in \{0.0, 0.1, 0.3, 0.7\}$
- **thread** = 18
- **lr** = 2
- **epoch** = 300

[5]Scikit-Learn random forest classifier implementation: `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier`

- **dim** = 30
- **wordNgrams** = 5

*Block-Walk Embedding*

This custom model takes inspiration from NLP and language modeling to create an embedding at the basic block level. Rather than using the raw bytes or disassembled tokens, we extract basic blocks using angr into a "vocabulary" analogous to how an NLP model builds a vocabulary of words. In preliminary experiments we noticed diminishing improvements when increasing the embedding dimension above 200 for both the block and walk embeddings layers.

Figure 4.4 shows this model architecture. To preprocess the data for this model, we extract basic blocks from each object file and normalize hex literals. After normalization, this dataset contains a vocabulary of about 18500 unique basic blocks. From these basic blocks we sample random walks of length 5 to construct a secondary set of sequences to encode the function's potential runtime control flows. We use PyTorch EmbeddingBag layers to create block and walk embeddings. Both EmbeddingBag layers use the vocabulary size for their number of embeddings and each uses `mode="max"` as its reduction function. Because the input to these EmbeddingBags are multiple sequences in flattened form, they also receive a positional encoding input containing the starting offset of each sequence.

The model is implemented using a sequence of PyTorch Modules with a total of 5203666 gradient-tuned parameters. When saved using the `torch.save` function, the trained model and its weights is about 20MB in size. Its layers have the following parameters:

- EmbeddingBag for basic blocks: **num_embeddings** = 18577, **embedding_dim** = 200, **mode** = 'max'
- EmbeddingBag for random walks: **num_embeddings** = 18577, **embedding_dim** = 80, **mode** = 'max'
- Dropout: **p** = 0.45, **inplace** = False

- LeakyReLU: **negative_slope** = 0.01
- AdaptiveMaxPool1d: **output_size** = 80
- Linear: **in_features** = 80, **out_features** = 26 (25 functions plus one "unknown" class), **bias** = True

We pass sequences of blocks as well as sequences of random walks to the model as inputs, but all sequences are not equal length. To inform the model of the size of each sequence, during preprocessing we compute a positional encoding for both blocks and walks. The number of examples given to the model in each case depends on the batch dimension which in turn varies depending on the training dataset used.

*Evaluation Environment*

All experiments were performed on an Ubuntu 22.04.3 LTS workstation with an 18-core Intel(R) Core(TM) i9-7980XE CPU@2.60GHz and 64GB of CPU memory. For GPU acceleration in PyTorch we use an Nvidia GeForce RTX 2060 (Revision A) with CUDA version 11.4 with 6GB of GPU memory. We used PyTorch version 2.0.1 and Scikit-Learn version 1.3.0 in the experiments described in this work.

### 4.3 Results and Discussion

In this section we evaluate the models and compare their results, with tables 4.4 through 4.9 showing macro-averaged F1 scores for different combinations of train and test data. In these tables the highest F1 score is highlighted in **bold**. Each table lists the size of the training set $N$ and test set $M$ for the given experiment. The "setup" column denotes one of the following combinations of train and test datasets:

1. O/O: train **Obfuscated**, test **Obfuscated** (Table 4.4)
2. O/P: train **Obfuscated**, test **Plain** (Table 4.5)

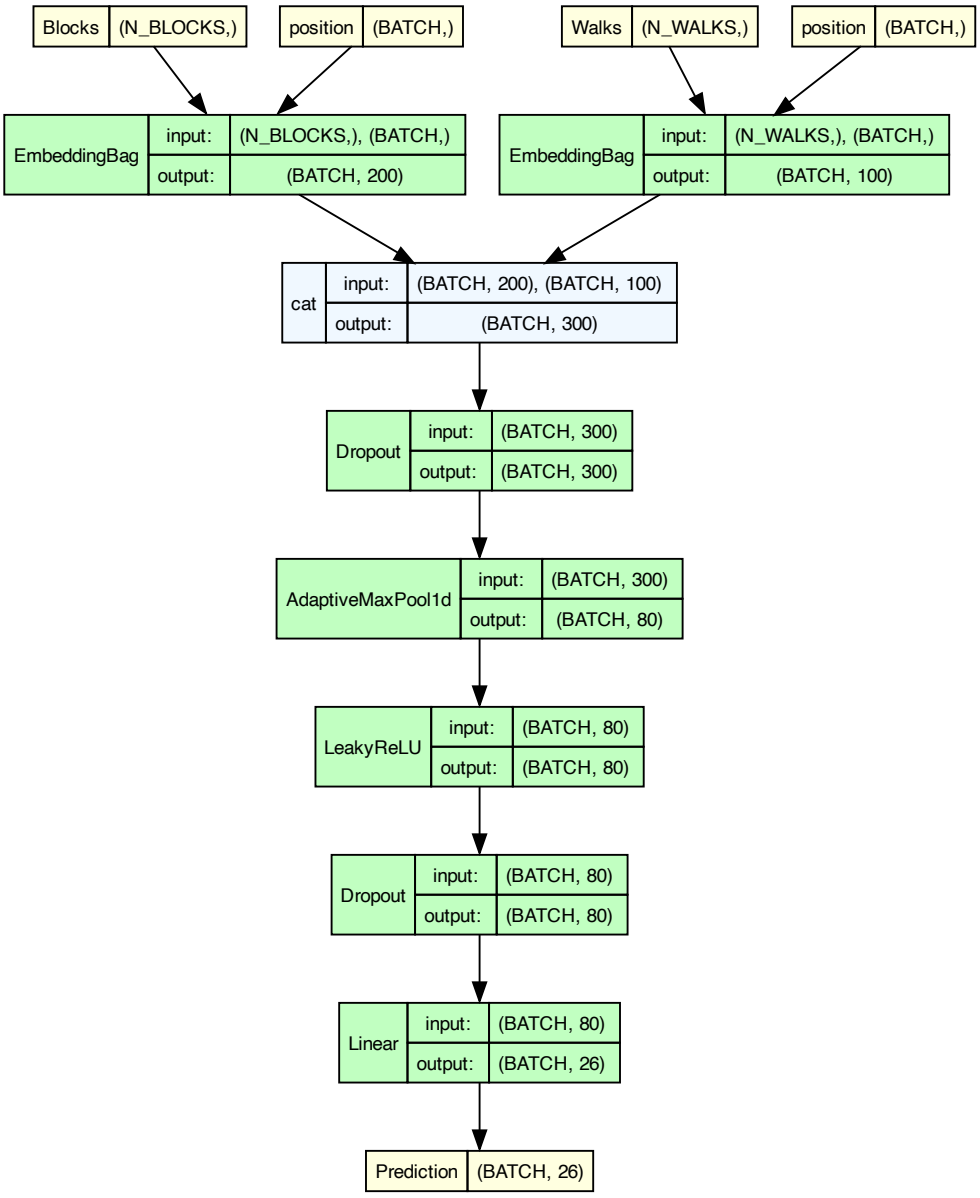



Figure 4.4: Block-Walk Embedding model; `BATCH` dimension depends on the batch size for each dataset

3. P/O: train **Plain**, test **Obfuscated** (Table 4.6)

4. P/P: train **Plain**, test **Plain** (Table 4.7)

5. S/O: train **Plain+Subset**, test **Obfuscated** (Table 4.8)

6. S/P: train **Plain+Subset**, test **Plain** (Table 4.9)

These tables can also help an analyst choose what model is best suited to their task. For example, the kNN model performs best in the scenario where the training data is **Obfuscated** and the test data is **Plain**. However, there is no distinction between training and inference in kNN, as each new prediction must compare with all previous measurements to determine its NCD value. This means the runtime for kNN has quadratic time complexity, and this is reflected in the test time column. If an application prefers faster predictions, kNN is typically the worst choice.

Another consideration is dataset size. The Random Forest model does best in the **Plain / Plain** scenario which has the smallest data sizes. In these evaluations we use a similar experimental setup and apply similar models to [71]. In particular, our experiment using **Obfuscated** data for both training and testing

Figure 4.5 shows detailed predicted-versus-actual values for all the functions in different dataset configurations.

### 4.3.1 Block-Walk Embedding

We train this model in using Stochastic Gradient Descent (SGD) with two portions of our dataset over 150 epochs, which takes about 23 seconds in our test environment. The initial learning rate is 20 with a learning rate decay scheduler[6] (factor=0.9).

For each epoch our model accepts one mini-batch of 256 obfuscated samples from the **Subset** portion of our dataset, and one mini-batch 64 **Plain** samples. For each of these mini-batches, we compute a separate Cross Entropy Loss and compute a weighted sum the two losses. In each

[6]Learning rate decay: `https://pytorch.org/docs/stable/generated/torch.optim.lr_scheduler.ReduceLROnPlateau.html`

Table 4.4: Macro-averaged F1 scores (train Obfuscated / test Obfuscated)

| model | parameter | setup | train(N) | test(N) | train(t) | test(t) | F1 |
|---|---|---|---|---|---|---|---|
| Block-Walk Embed | | O/O | 800 | 200 | 7.41 | 0.05 | 0.897 |
| kNN | neighbors=1 | O/O | 800 | 200 | 0.00 | 5.15 | 0.887 |
| kNN | neighbors=3 | O/O | 800 | 200 | 0.00 | 5.15 | 0.824 |
| kNN | neighbors=5 | O/O | 800 | 200 | 0.00 | 5.15 | 0.799 |
| kNN | neighbors=9 | O/O | 800 | 200 | 0.00 | 5.15 | 0.729 |
| RandomForest | estimators=40 | O/O | 800 | 200 | 0.07 | 0.01 | 0.895 |
| RandomForest | estimators=80 | O/O | 800 | 200 | 0.15 | 0.03 | **0.909** |
| RandomForest | estimators=100 | O/O | 800 | 200 | 0.17 | 0.03 | 0.894 |
| RandomForest | estimators=120 | O/O | 800 | 200 | 0.19 | 0.03 | 0.905 |
| $\text{RandomForest}_{\text{token}}$ | estimators=40 | O/O | 800 | 200 | 0.08 | 0.02 | 0.513 |
| $\text{RandomForest}_{\text{token}}$ | estimators=80 | O/O | 800 | 200 | 0.13 | 0.03 | 0.481 |
| $\text{RandomForest}_{\text{token}}$ | estimators=100 | O/O | 800 | 200 | 0.17 | 0.03 | 0.500 |
| $\text{RandomForest}_{\text{token}}$ | estimators=120 | O/O | 800 | 200 | 0.19 | 0.03 | 0.494 |
| FastText | threshold=0.0 | O/O | 800 | 200 | 6.08 | 0.06 | 0.430 |
| FastText | threshold=0.1 | O/O | 800 | 200 | 6.08 | 0.03 | 0.430 |
| FastText | threshold=0.3 | O/O | 800 | 200 | 6.08 | 0.02 | 0.443 |
| FastText | threshold=0.7 | O/O | 800 | 200 | 6.08 | 0.02 | 0.343 |

Table 4.5: Macro-averaged F1 scores (train Plain / test Obfuscated)

| model | parameter | setup | train(N) | test(N) | train(t) | test(t) | F1 |
|---|---|---|---|---|---|---|---|
| Block-Walk Embed | | P/O | 95 | 200 | 1.71 | 0.05 | **0.637** |
| kNN | neighbors=1 | P/O | 95 | 200 | 0.00 | 0.75 | 0.623 |
| kNN | neighbors=3 | P/O | 95 | 200 | 0.00 | 0.75 | 0.555 |
| kNN | neighbors=5 | P/O | 95 | 200 | 0.00 | 0.75 | 0.499 |
| kNN | neighbors=9 | P/O | 95 | 200 | 0.00 | 0.75 | 0.479 |
| RandomForest | estimators=40 | P/O | 95 | 200 | 0.07 | 0.02 | 0.523 |
| RandomForest | estimators=80 | P/O | 95 | 200 | 0.14 | 0.03 | 0.530 |
| RandomForest | estimators=100 | P/O | 95 | 200 | 0.19 | 0.03 | 0.606 |
| RandomForest | estimators=120 | P/O | 95 | 200 | 0.18 | 0.03 | 0.613 |
| $\text{RandomForest}_{\text{token}}$ | estimators=40 | P/O | 95 | 200 | 0.07 | 0.02 | 0.174 |
| $\text{RandomForest}_{\text{token}}$ | estimators=80 | P/O | 95 | 200 | 0.13 | 0.03 | 0.177 |
| $\text{RandomForest}_{\text{token}}$ | estimators=100 | P/O | 95 | 200 | 0.15 | 0.03 | 0.185 |
| $\text{RandomForest}_{\text{token}}$ | estimators=120 | P/O | 95 | 200 | 0.20 | 0.03 | 0.176 |
| FastText | threshold=0.0 | P/O | 95 | 200 | 0.75 | 0.06 | 0.187 |
| FastText | threshold=0.1 | P/O | 95 | 200 | 0.75 | 0.03 | 0.187 |
| FastText | threshold=0.3 | P/O | 95 | 200 | 0.75 | 0.02 | 0.181 |
| FastText | threshold=0.7 | P/O | 95 | 200 | 0.75 | 0.02 | 0.049 |

Table 4.6: Macro-averaged F1 scores (train Subset / test Obfuscated)

| model | parameter | setup | train(N) | test(N) | train(t) | test(t) | F1 |
|---|---|---|---|---|---|---|---|
| Block-Walk Embed | | S/O | 800 | 200 | 4.62 | 0.05 | **0.798** |
| kNN | neighbors=1 | S/O | 800 | 200 | 0.00 | 5.36 | 0.316 |
| kNN | neighbors=3 | S/O | 800 | 200 | 0.00 | 5.36 | 0.312 |
| kNN | neighbors=5 | S/O | 800 | 200 | 0.00 | 5.36 | 0.310 |
| kNN | neighbors=9 | S/O | 800 | 200 | 0.00 | 5.36 | 0.307 |
| RandomForest | estimators=40 | S/O | 800 | 200 | 0.08 | 0.02 | 0.316 |
| RandomForest | estimators=80 | S/O | 800 | 200 | 0.15 | 0.03 | 0.324 |
| RandomForest | estimators=100 | S/O | 800 | 200 | 0.16 | 0.03 | 0.335 |
| RandomForest | estimators=120 | S/O | 800 | 200 | 0.19 | 0.03 | 0.332 |
| $\text{RandomForest}_{\text{token}}$ | estimators=40 | S/O | 800 | 200 | 0.09 | 0.02 | 0.203 |
| $\text{RandomForest}_{\text{token}}$ | estimators=80 | S/O | 800 | 200 | 0.15 | 0.03 | 0.206 |
| $\text{RandomForest}_{\text{token}}$ | estimators=100 | S/O | 800 | 200 | 0.18 | 0.03 | 0.204 |
| $\text{RandomForest}_{\text{token}}$ | estimators=120 | S/O | 800 | 200 | 0.19 | 0.03 | 0.202 |
| FastText | threshold=0.0 | S/O | 800 | 200 | 5.88 | 0.06 | 0.208 |
| FastText | threshold=0.1 | S/O | 800 | 200 | 5.88 | 0.03 | 0.208 |
| FastText | threshold=0.3 | S/O | 800 | 200 | 5.88 | 0.02 | 0.219 |
| FastText | threshold=0.7 | S/O | 800 | 200 | 5.88 | 0.02 | 0.182 |

Table 4.7: Macro-averaged F1 scores (train Obfuscated / test Plain)

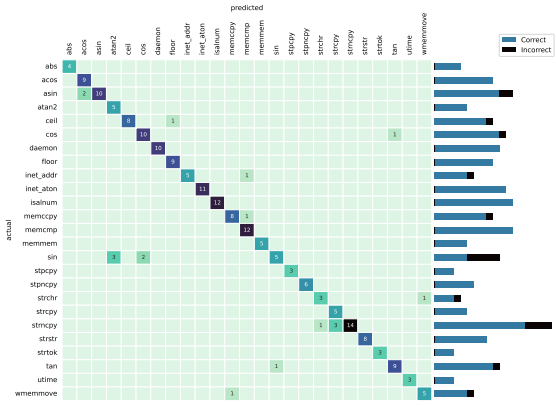
| model | parameter | setup | train(N) | test(N) | train(t) | test(t) | F1 |
|---|---|---|---|---|---|---|---|
| Block-Walk Embed | | O/P | 800 | 43 | 7.41 | 0.01 | 0.400 |
| kNN | neighbors=1 | O/P | 800 | 43 | 0.00 | 2.73 | **0.868** |
| kNN | neighbors=3 | O/P | 800 | 43 | 0.00 | 2.73 | 0.753 |
| kNN | neighbors=5 | O/P | 800 | 43 | 0.00 | 2.73 | 0.650 |
| kNN | neighbors=9 | O/P | 800 | 43 | 0.00 | 2.73 | 0.554 |
| RandomForest | estimators=40 | O/P | 800 | 43 | 0.08 | 0.02 | 0.850 |
| RandomForest | estimators=80 | O/P | 800 | 43 | 0.13 | 0.01 | 0.850 |
| RandomForest | estimators=100 | O/P | 800 | 43 | 0.17 | 0.03 | 0.819 |
| RandomForest | estimators=120 | O/P | 800 | 43 | 0.20 | 0.03 | 0.819 |
| $\text{RandomForest}_{\text{token}}$ | estimators=40 | O/P | 800 | 43 | 0.08 | 0.02 | 0.413 |
| $\text{RandomForest}_{\text{token}}$ | estimators=80 | O/P | 800 | 43 | 0.14 | 0.03 | 0.409 |
| $\text{RandomForest}_{\text{token}}$ | estimators=100 | O/P | 800 | 43 | 0.17 | 0.03 | 0.405 |
| $\text{RandomForest}_{\text{token}}$ | estimators=120 | O/P | 800 | 43 | 0.20 | 0.03 | 0.409 |
| FastText | threshold=0.0 | O/P | 800 | 43 | 6.08 | 0.01 | 0.482 |
| FastText | threshold=0.1 | O/P | 800 | 43 | 6.08 | 0.01 | 0.482 |
| FastText | threshold=0.3 | O/P | 800 | 43 | 6.08 | 0.01 | 0.485 |
| FastText | threshold=0.7 | O/P | 800 | 43 | 6.08 | 0.01 | 0.271 |

Table 4.8: Macro-averaged F1 scores (train Plain / test Plain)

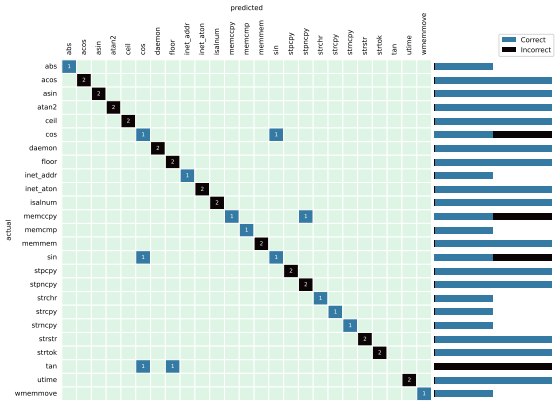
| model | parameter | setup | train(N) | test(N) | train(t) | test(t) | F1 |
|---|---|---|---|---|---|---|---|
| Block-Walk Embed | | P/P | 95 | 43 | 1.75 | 0.01 | 0.299 |
| kNN | neighbors=1 | P/P | 95 | 43 | 0.00 | 0.40 | 0.627 |
| kNN | neighbors=3 | P/P | 95 | 43 | 0.00 | 0.40 | 0.472 |
| kNN | neighbors=5 | P/P | 95 | 43 | 0.00 | 0.40 | 0.390 |
| kNN | neighbors=9 | P/P | 95 | 43 | 0.00 | 0.40 | 0.200 |
| RandomForest | estimators=40 | P/P | 95 | 43 | 0.06 | 0.01 | **0.655** |
| RandomForest | estimators=80 | P/P | 95 | 43 | 0.16 | 0.02 | 0.565 |
| RandomForest | estimators=100 | P/P | 95 | 43 | 0.15 | 0.03 | 0.653 |
| RandomForest | estimators=120 | P/P | 95 | 43 | 0.18 | 0.03 | 0.641 |
| $\text{RandomForest}_{\text{token}}$ | estimators=40 | P/P | 95 | 43 | 0.07 | 0.02 | 0.347 |
| $\text{RandomForest}_{\text{token}}$ | estimators=80 | P/P | 95 | 43 | 0.12 | 0.02 | 0.368 |
| $\text{RandomForest}_{\text{token}}$ | estimators=100 | P/P | 95 | 43 | 0.14 | 0.03 | 0.368 |
| $\text{RandomForest}_{\text{token}}$ | estimators=120 | P/P | 95 | 43 | 0.18 | 0.03 | 0.354 |
| FastText | threshold=0.0 | P/P | 95 | 43 | 0.75 | 0.01 | 0.387 |
| FastText | threshold=0.1 | P/P | 95 | 43 | 0.75 | 0.01 | 0.387 |
| FastText | threshold=0.3 | P/P | 95 | 43 | 0.75 | 0.01 | 0.381 |
| FastText | threshold=0.7 | P/P | 95 | 43 | 0.75 | 0.01 | 0.163 |

Table 4.9: Macro-averaged F1 scores (train Subset / test Plain)

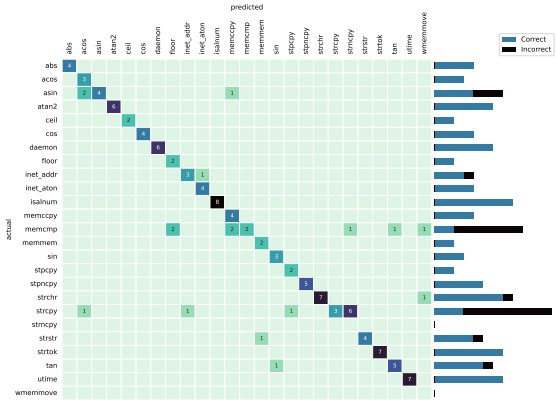
| model | parameter | setup | train(N) | test(N) | train(t) | test(t) | F1 |
|---|---|---|---|---|---|---|---|
| Block-Walk Embed | | S/P | 800 | 43 | 3.69 | 0.01 | **0.400** |
| kNN | neighbors=1 | S/P | 800 | 43 | 0.00 | 2.87 | 0.368 |
| kNN | neighbors=3 | S/P | 800 | 43 | 0.00 | 2.87 | 0.338 |
| kNN | neighbors=5 | S/P | 800 | 43 | 0.00 | 2.87 | 0.321 |
| kNN | neighbors=9 | S/P | 800 | 43 | 0.00 | 2.87 | 0.298 |
| RandomForest | estimators=40 | S/P | 800 | 43 | 0.07 | 0.01 | 0.368 |
| RandomForest | estimators=80 | S/P | 800 | 43 | 0.17 | 0.03 | 0.391 |
| RandomForest | estimators=100 | S/P | 800 | 43 | 0.17 | 0.03 | 0.383 |
| RandomForest | estimators=120 | S/P | 800 | 43 | 0.20 | 0.03 | 0.391 |
| $\text{RandomForest}_{\text{token}}$ | estimators=40 | S/P | 800 | 43 | 0.09 | 0.02 | 0.234 |
| $\text{RandomForest}_{\text{token}}$ | estimators=80 | S/P | 800 | 43 | 0.14 | 0.02 | 0.234 |
| $\text{RandomForest}_{\text{token}}$ | estimators=100 | S/P | 800 | 43 | 0.18 | 0.03 | 0.228 |
| $\text{RandomForest}_{\text{token}}$ | estimators=120 | S/P | 800 | 43 | 0.20 | 0.03 | 0.229 |
| FastText | threshold=0.0 | S/P | 800 | 43 | 5.88 | 0.01 | 0.209 |
| FastText | threshold=0.1 | S/P | 800 | 43 | 5.88 | 0.01 | 0.209 |
| FastText | threshold=0.3 | S/P | 800 | 43 | 5.88 | 0.01 | 0.235 |
| FastText | threshold=0.7 | S/P | 800 | 43 | 5.88 | 0.01 | 0.231 |



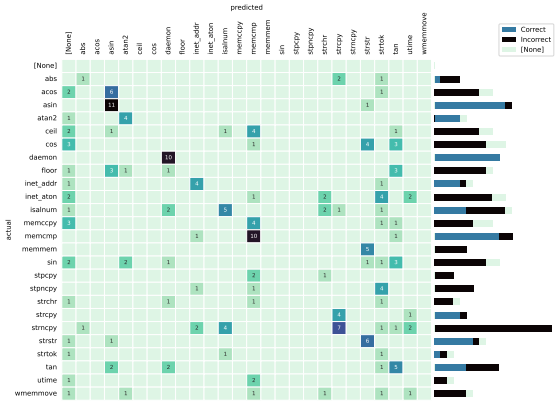

Random Forest (best model in O/O)




kNN (best model in O/P)




Block-Walk Embedding (best model in S/O)




FastText (worst model in S/O)

Figure 4.5: Confusion matrices for different models and train/test configurations

training iteration, we add the computed **Plain** and **Subset** loss with equal weight before updating gradients as per Equation 4.4. Here we extract the weighting parameter $\alpha$ because in principle these different training data could be weighted separately so long as their weights sum to 1. We experimented with different weights but found no significant differences from $\alpha = 0.3$ to $\alpha = 0.7$.

$$\begin{aligned} \alpha &= 0.5 \\ \text{loss} &= \alpha \times \text{loss}_{Plain} + (1 - \alpha) \times \text{loss}_{Subset} \end{aligned} \tag{4.4}$$

We use an initial learning rate of 20, decreased over the course of training using PyTorch's `ReduceLROnPlateau` learning rate scheduler which decays the learning rate by a factor of 0.9 whenever the validation accuracy in the current epoch is lower than the previous epoch. The

loss, training accuracy, and validation accuracy are shown for this training in Figure 4.6.

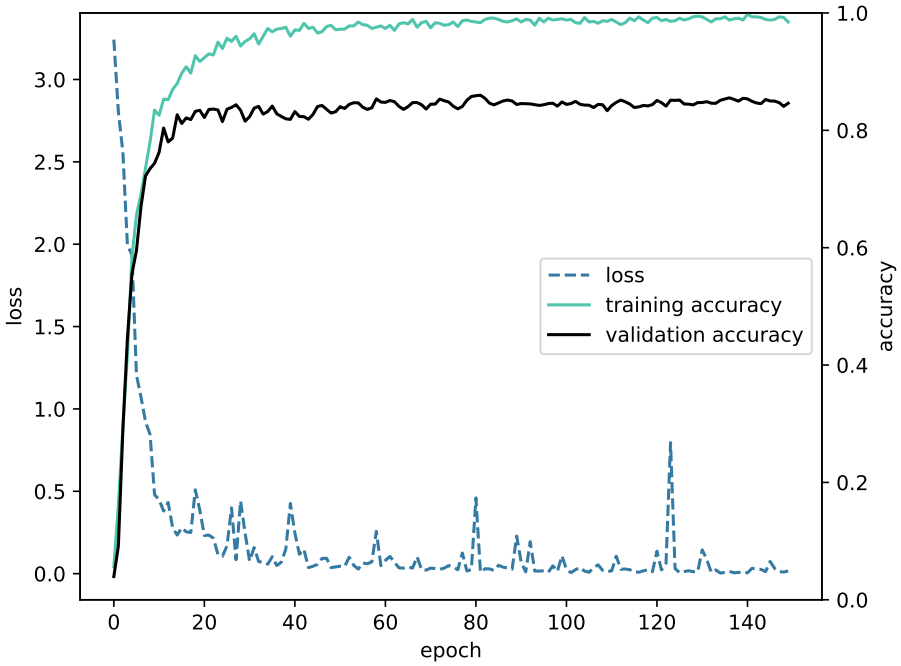



Figure 4.6: Training the Block-Walk Embedding model

Some **Plain** functions do not appear in their respective test set and we give them an F1 score of 0 by default. We include individual function F1 scores as well as macro- and micro-averaged F1 scores for the model as a whole in Table 4.10. When tested with a large enough dataset, we find that combined training improves F1 scores for most functions and offers significant overall improvements. For example, the `sin` function is absent from the Subset training data, but by learning about `sin` from the Plain data and also learning about obfuscation from the rest of the Subset data, the model is better able to detect obfuscated examples of `sin`.

### 4.3.2 Function Detection

Finding a known function within an unknown binary file is a reverse engineering task used in the contexts of malware identification and intellectual property protection. In practice, most reverse engineering practitioners employ sophisticated software suites such as IDA Pro or Ghidra, which contain disassemblers, decompilers, visualization tools, and multiple analysis workflows. De-

Table 4.10: F1 scores per-function after 150 training epochs

| $f$ | O/O | P/O | S/O | O/P | P/P | S/P | $f \in$ S |
|---|---|---|---|---|---|---|---|
| abs | 1.00 | 0.40 | 1.00 | 0.00 | 0.00 | 0.00 | yes |
| acos | 0.67 | 0.73 | 0.67 | 0.00 | 0.00 | 0.00 | no |
| asin | 0.57 | 0.75 | 0.80 | 1.00 | 0.67 | 1.00 | yes |
| atan2 | 1.00 | 0.75 | 1.00 | 1.00 | 1.00 | 1.00 | yes |
| ceil | 0.80 | 0.80 | 0.80 | 0.00 | 0.00 | 0.00 | no |
| cos | 0.86 | 0.89 | 1.00 | 0.00 | 0.00 | 0.00 | no |
| daemon | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | yes |
| floor | 0.86 | 0.86 | 0.67 | 0.00 | 0.00 | 0.00 | no |
| inet_addr | 1.00 | 0.20 | 0.75 | 1.00 | 1.00 | 1.00 | yes |
| inet_aton | 1.00 | 0.89 | 0.89 | 1.00 | 0.00 | 1.00 | no |
| isalnum | 1.00 | 0.80 | 1.00 | 0.00 | 0.00 | 0.00 | yes |
| memccpy | 1.00 | 0.83 | 0.77 | 0.00 | 0.00 | 0.00 | no |
| memcmp | 0.67 | 0.25 | 0.50 | 1.00 | 1.00 | 1.00 | yes |
| memmem | 1.00 | 0.80 | 0.80 | 0.00 | 0.00 | 0.00 | no |
| sin | 0.73 | 0.80 | 1.00 | 1.00 | 0.80 | 1.00 | no |
| stpcpy | 1.00 | 0.44 | 0.80 | 1.00 | 1.00 | 1.00 | no |
| stpncpy | 0.91 | 0.89 | 0.89 | 0.00 | 0.00 | 0.00 | no |
| strchr | 1.00 | 0.18 | 0.93 | 0.00 | 0.00 | 0.00 | yes |
| strcpy | 0.86 | 0.50 | 0.50 | 0.00 | 0.00 | 0.00 | yes |
| strncpy | 0.77 | 0.57 | 0.44 | 0.00 | 0.00 | 0.00 | no |
| strstr | 1.00 | 0.89 | 0.89 | 0.00 | 0.00 | 0.00 | yes |
| strtok | 0.93 | 0.44 | 1.00 | 1.00 | 1.00 | 1.00 | yes |
| tan | 0.80 | 0.50 | 0.92 | 1.00 | 0.00 | 1.00 | yes |
| utime | 1.00 | 0.43 | 0.93 | 0.00 | 0.00 | 0.00 | yes |
| wmemmove | 1.00 | 0.33 | 0.00 | 0.00 | 0.00 | 0.00 | no |
| F1 (macro) | 0.89 | 0.67 | 0.80 | 0.40 | 0.29 | 0.40 | |
| F1 (micro) | 0.91 | 0.65 | 0.84 | 0.92 | 0.69 | 0.92 | |

spite this sophistication, reverse engineering is generally labor-intensive and requires specialized knowledge and experience. One aim of our work is to improve the day-to-day productivity of such practitioners. Toward that goal, our obfuscated-based approach is designed to detect similar but not identical examples of known functions. By making a prediction that some part of an unknown binary is most similar to one known function or another, we can help the reverse engineer hone in on more interesting regions of the binary, and spend less time reverse engineering something that is already known or is uninteresting from an analysis perspective.

To evaluate our model on the real-world task of detecting similar functions within an unknown binary, we select open-source samples of statically linked executable files[7] and malware samples[8]. These programs are stripped of debug symbols, but because the source code for some of the samples is available, we can check the quality of our model's predictions.

Our methods trained on binary data of varying sizes, but each of our samples contained only one logical function. Real world programs contain multiple functions, so instead of reporting which of the training functions is most similar to the *entire* test program, we instead consider partial context windows within the larger test program. The choice of context size is somewhat arbitrary, but in the interests of pragmatism we choose a context window large enough to contain 70% of our obfuscated training samples.

As we move the context window in overlapping strides across the test program, the model predicts a similarity score between the training data and the subset of the test program contained by the window. We convert this score to probabilities and exclude any prediction below some threshold $T$ within a context window size $W$ moving with stride size $S$.

In Figure 4.7, we use the Block-Walk Embedding model to analyze a single malware file corresponding to the Petya ransomware [78]. The results of applying our model to unlabeled data such as this malware sample require a degree of interpretation and skepticism. Our training data is an arbitrary selection of standard library functions, so what does it mean for our model to classify part of a malware file as "similar" to a training class? One possibility is that the unknown binary ac-

[7]Static-binaries repository: `https://github.com/andrew-d/static-binaries`

[8]"the Zoo" malware repository: `https://github.com/ytisf/theZoo/`

tually contains the predicted library function. On the other hand, a more likely explanation is that this section of the malware simply shares some characteristics to the training function. However, if a section of the malware is a close match for a function such as `ceil` that may be less interesting than if a section of malware is a close match for `inet_aton`, because that more strongly indicates the malware is trying to connect to a command and control server or exfiltrating data.

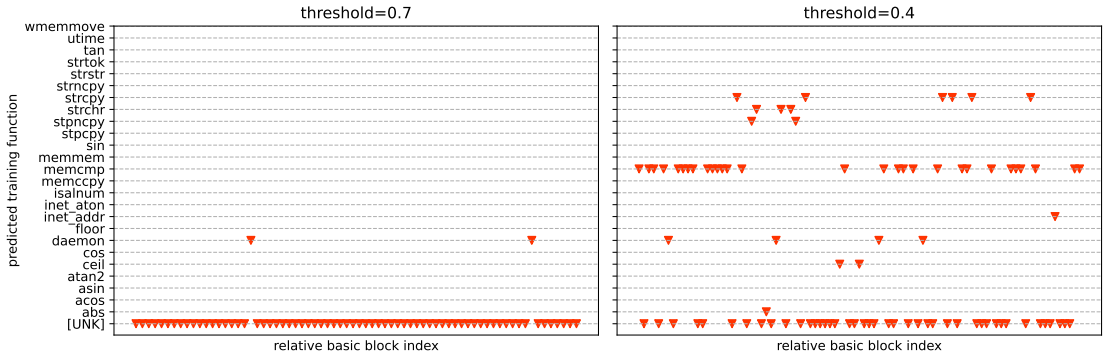



Figure 4.7: Function predictions in "petya0" binary

In Figure 4.8, we show multiple benign and malware programs side-by-side. The benign programs are from an open-source collection of statically compiled binaries. Interpreting results for benign programs requires the same degree of judgment as for malware programs, but we can notice some differences. For example, two different versions of petya (petya0/petya1) have more in common with each other than with the wannacry malware. Meanwhile, all the benign programs have some differences from the malware programs.

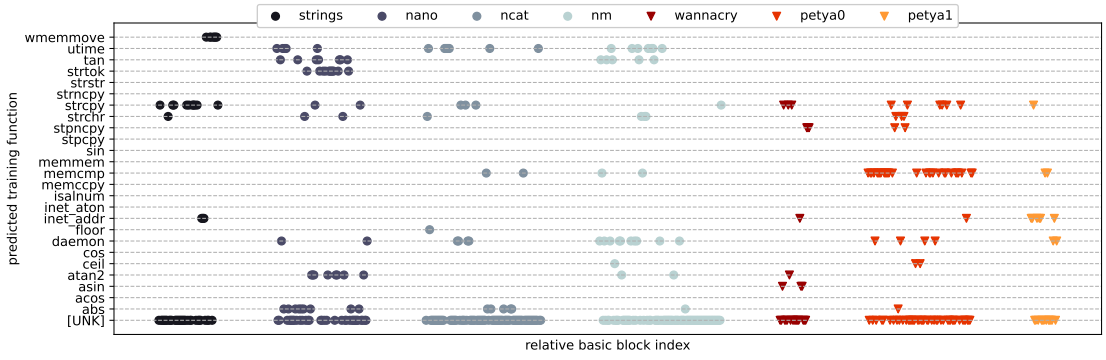



Figure 4.8: Function predictions in assorted programs ($T = 0.7, W = 30, S = 20$)

## 4.4 Conclusions

In this work we propose a method of data augmentation for binary similarity analysis based on source code obfuscation. We demonstrate different models capable of detecting similar functions with a training set with varying degrees of feature engineering. Our results show that it is possible for these models to generalize to unseen test data and learn different features from both **Plain** and **Obfuscated** datasets. Our Block-Walk Embedding model takes advantage of the graph structure inherent in most compiled binary code and analyzes binaries at the basic block level of granularity. Extracting basic blocks depends on additional feature engineering beyond the level of a disassembler. In our work, we used the angr analysis framework, but similar facilities exist in IDA Pro, Ghidra, Radare2[9], and other reverse engineering software. For our purposes, angr was preferable for this task due to its runtime speed and ease of integration with existing Python-based scripts.

While the motivation for this work is to determine semantics for an unknown binary, in practice this is an intractable problem. But by reducing this problem to one of approximate similarity to a set of known functions, we can arrive at a practical implementation. Future works should note that the most effective use of our approach is to treat the training functions as focal points. With a model trained to recognize a small set of "interesting" functions, a reverse engineer can analyze an unknown binary more effectively. Large swaths of the unknown binary may be uninteresting for analysis purposes, but the engineer can target their efforts on those regions that contain close matches.

One limitation of this work is that in order to train on highly obfuscated code, an advanced obfuscation framework is required. Tigress satisfies this requirement, but comes with additional requirements that the original source code must be written in the C programming language, and the analyst must have a copy of this source code. Similar approaches in this domain often utilize obfuscator-llvm [26] instead of Tigress, which operates on llvm's intermediate representation (IR) rather than on C source code. While currently obfuscator-llvm has a small set of transformations, it inherently supports multiple programming languages and is a cross-compiler, which greatly sim-

[9]Radare2 reverse engineering software: `https://www.radare.org/r/`

plifies the task of exploring obfuscation for different platforms and programming environments. A more significant limitation is this work's focus on obfuscation types that alter literal values or control flow. Meanwhile, more advanced obfuscation techniques exist. Virtualization, in which the code is obfuscated by implementing a virtual machine and then encoding instructions for that machine, is a more challenging transformation to analyze due to the higher-order representation. Similarly, just-in-time (JIT) compilation is a transformation that occurs at runtime. Because our method uses purely static analysis, it may be less effective with JIT or virtualized transformations. In this work we did not study these because while Tigress includes such transformation options, we found that these required significant manual work to produce valid executables. Some of these limitations are due to Tigress' underlying C parsing engine not supporting newer C standard features, but it is an area which future work could explore.

As a practical consideration, obtaining source code for certain types of programs is not always possible, but when it is, we propose making the most of it with our obfuscation-based data augmentation method, and training on a mixture of obfuscated and plain data sources. This should promote models to learn about the ways code can be obfuscated while also specializing on the functions of interest to the reverse engineer.

# CHAPTER 5

# RELATED WORK

## 5.1 Introduction

Today's engineers use complex tools to transform their ideas and specifications into binary artifacts. The tools for constructing hardware designs and software designs differ in important ways, but they also share fundamental similarities. Most notably, both hardware and software construction depends on multiple transformations to source code. Many of these transformations are performed on intermediate, graph representations of the source code's syntax and semantics.

These tools make up the lowest levels of the hardware and software supply chain. Everything built above this level inherits vulnerabilities of the lower levels [79], making these tools attractive targets for attackers. While mitigations have been proposed for software [80], similar fixes may be required for hardware synthesis tools.

Open source projects like LLVM and GCC have advanced the state the art of compilers, and reduced the barrier to entry for security analysts. This is one reason why commercial companies contribute to these projects, even though their contributions may help their competitors. Concentration of effort allows more specialization, and gives niche research a path to practical implementation, by integrating with an existing project. Features that were once restricted to academic or special-purpose compilers are now accessible in mainstream compilers A similar open source revolution has yet to occur for hardware synthesis tools, in part because the diversity of hardware makes standardization difficult.

### 5.1.1 Threats Affecting Software and Hardware

Despite the differences between compilers and hardware synthesis tools, and despite differences in approach (open source versus proprietary), hardware and software systems face similar threats. The most relevant threats are defined below.

**Piracy** When a company's competitor copies their software to sell it as their own, a form of intellectual property (IP) theft.

**Malware** Software that either harms the legitimate user, or subverts their machine for the attacker's purposes.

**Ransomware** A form of malware which encrypts the user's files and demands payment in exchange for an unlocking key.

**Trojan** A type of malware which infiltrates the target's system in order to gain control. This type of attack is sometimes the entry point for other attacks, such as ransomware.

**Side Channel Attack** Unauthorized access to a system by indirect means. This attack can take many forms, including Van Eck phreaking [81], detecting keystrokes through motion analysis [82], bypassing encryption by monitoring power consumption [83], leaking data by exploiting DRAM refresh cycles [84], or bypassing address space layout randomization by monitoring the timing of a memory management unit [85].

### 5.1.2 Vulnerabilities Are Bugs

While it may be true that the malware author considers vulnerabilities to be *features*, for the rest of us they are *bugs*. Error mitigation strategies for both software and hardware have much in common, such as the use of functional and behavioral testing. However, testing can only prove the presence of bugs, not their absence [86]. Analysis of initial source code can catch some bugs, but ultimately the compiled or synthesized binary artifacts should also be analyzed, in case additional errors are introduced by the compiler or synthesis toolchains. Behavioral testing (or "functional" or "integration" testing) is a common way to test the compiled binary file, since this ideally exercises the same code which would be distributed to end-users.

In software, many aspects of a program can be encoded in the programming language's type system. Programmers can leverage *strengths* of type systems to avoid entire classes of bugs at compile time, with the automated error checking provided by the compiler and the type system. For

example, the Rust programming language enforces variable ownership and lifetimes in its type system, preventing use-after-free errors which are common to languages like C and C++ which use manual memory management. Notably, Rust's type system also prevents data races which are common even in garbage-collected languages like Java and Go [87]. In addition to type system guarantees, programming languages can take advantage of other forms of static analysis, in part because compilers permit analysis of intermediate representations, before transforming the code to hardware-specific binary formats. Conversely, *weaknesses* in a type system can become "billion dollar mistakes" as in the case of null references [88]. Finally, dynamic or run time testing is also used for programming languages, and again software has an advantage because of the relatively small number of hardware platforms they must support. These hardware platforms can be emulated, allowing programmers to test their code more cheaply on virtual hardware.

In hardware, functional tests are ubiquitously used to help ensure correctness. The drawback to testing in hardware is that the platform is entirely or mostly customized to the application, which means there is no standard for emulation. This does not prevent emulation in theory, but in practice it means that emulators for hardware are one-off designs that can not be reused for other applications. Tests are especially important in hardware because commonly used hardware description languages like Verilog and VHDL have only a rudimentary type system able to distinguish between registers and wires of different sizes.

A further limitation of error checking for hardware is the lack of standardized intermediate representation. This lack ensures that hardware synthesis tools remain self-contained and unable to benefit from related work on aspects of the transformation process. For example, if hardware semantics at the register-transfer level were standardized by all vendors, then Verilog written in a Xilinx tool could be more easily ported to an Altera FPGA, and vice versa. Standard interfaces also facilitate emulation and cheaper testing, which encourages more types of testing, which in turn can catch more types of errors. However, recent progress in embedded system peripheral emulation suggests a way forward for generic hardware emulation [89].

### 5.1.3 Software Compilation

While software comes in many forms, this work focuses on *compiled* software. A *compiler* is a program which transforms human-written source code into binary executable code. This binary format is essentially unreadable by humans, which hinders analysis. This is especially true when the software is distributed in binary form only.

Given this difficulty, why would anyone analyze the binary instead of the source code? Because the source code may not reflect the true meaning of the produced binary, due to compiler errors, undefined behavior, or malware. Unfortunately, disassembler and decompiler tools can never perfectly reverse the assembly and compilation processes, because those processes are typically "lossy" and discard some of the original source's information.

## 5.2 Motivation

As more aspects of our work, lives, society, and physical world become Internet connected, we gain both convenience and vulnerabilities. Recent supply-chain attacks demonstrate that these vulnerabilities have not only monetary or business consequences, but also physical impacts. For example, in early 2021, ransomware attacks [90], [91] impacted the US fuel and food supply. Attacks on these parts of our supply chains make headlines, but lower-level parts of the hardware and software supply chains have even wider scope and are therefore even more important to secure against threats. Malware is increasing over time, as shown in Figure 5.1.

### 5.2.1 Ransomware

Ransomware is a software version of extortion. When successful, it encrypts the target machine so the owners do not have access to their own data. The attacker holds the private key and can access the data at will. Attackers may then demand a ransom in exchange for the key, and may use "double extortion" and threaten to release private information. This private information can be customer information like credit cards, or proprietary information which would benefit competitors,

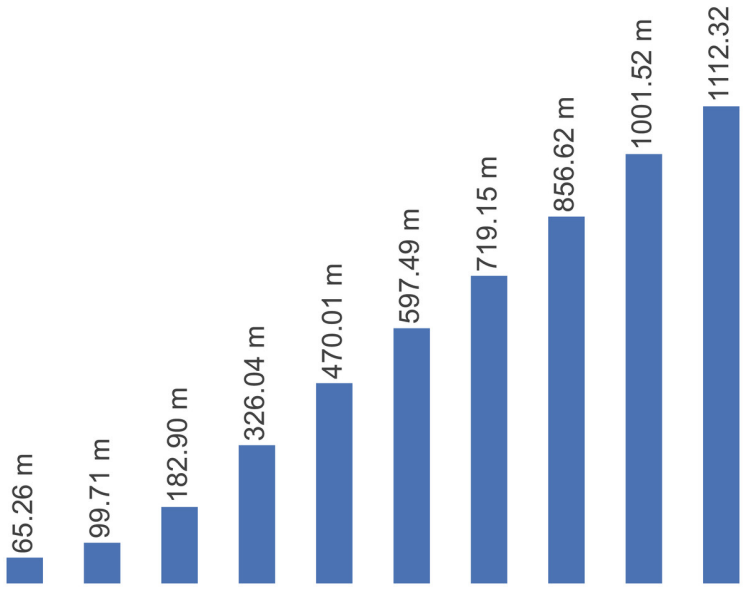



Figure 5.1: New Malware Identification 2011-2020 [7]

or information which would embarrass the people or company if it became public.

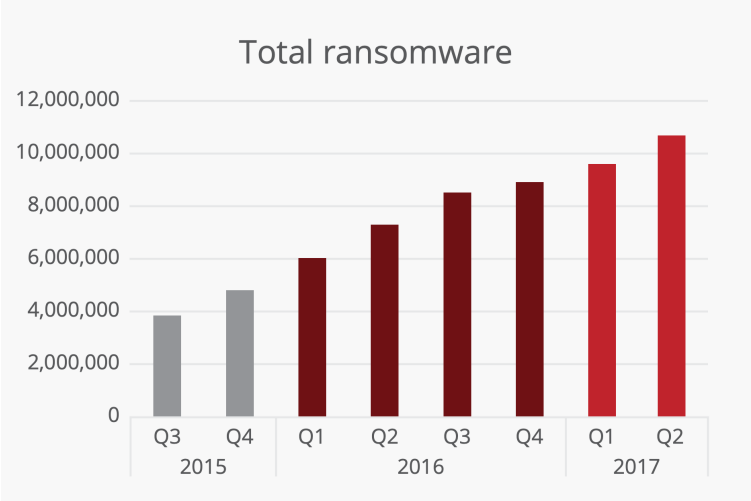



Figure 5.2: Ransomware Increasing [92]

As shown in Figure 5.2, ransomware has become increasingly popular in recent years, and is now a profitable criminal industry. There are software-as-a-service (SaaS) vendors of ransomware and multiple products geared toward this activity. Often, ransomware attacks appear to operate out of different countries [93] from where the attack takes place, taking advantage of extradition laws to reduce their own risk.

### 5.2.2 Cybersecurity

The 2020 data breach of the US federal government [94] utilized vulnerabilities in supply chain components involved in federated authentication. This attack leveraged vulnerabilities in multiple software components, at different levels in the software supply chain. More recently, researchers have noted [95] that the widespread use of commercial, off-the-shelf components has caused attackers to shift their focus earlier in the product lifecycle.

Connecting infrastructure to the Internet affords many advantages. For example, the recent COVID-19 global pandemic forced many businesses to switch to remote work for all but essential personnel. Connecting existing infrastructure to the Internet inherently increases its "attack surface". In addition, network connectivity software can bring its own vulnerabilities and bugs. But most importantly, because network connectivity is often added on to an existing system, the threat model of the original system may not be compatible with Internet connectivity, and so may expose additional vulnerabilities due to insecure configuration.

The concept of information security is broader than cybersecurity, as it includes topics such as state secrets, intellectual property, and personal information. But for the purposes of this work, there are two primary aspects of information security: information at rest, and information in transit. Information at rest includes all persistent storage of data. This can be stored on a hard drive, or on paper. Information in transit is when some data moves from one location to another, usually for the purposes of sharing the information between authorized parties.

Cryptography is about securing communications so that only those with authorization may understand the messages. In practice, this involves encrypting messages such that an eavesdropper would not be able to read the messages, or to prevent third parties from tampering with messages. End-to-end encryption is a concept in which messages are encrypted at all stages of the communication life-cycle - when they are generated, when they are in transit, and when they are read by the recipient.

One practical limitation of information security are illustrated by in Figure 5.3. Decryption is hard, but forcing a person to reveal the message is easier. There are internal threats to informa-

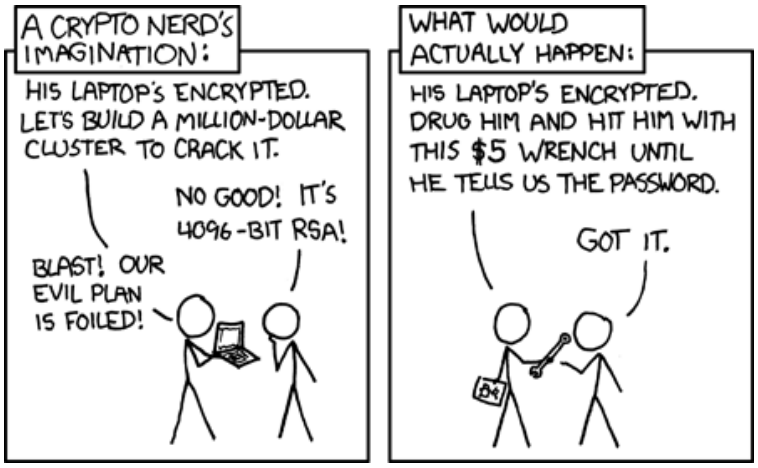



Figure 5.3: Threat Modeling [96]

tion security, such as employees who sell secrets to their employer's competitors. There are also external threats, such as competitors or even nation-states practicing industrial espionage. Despite best information security practices, malicious actors still exist. But instead of giving up in despair, practitioners use "threat models" to help decide appropriate steps.

The concept of a "threat model" implies imagining a way in which the service or information could be attacked, and then devise a protection against that specific attack. The reason for this narrow focus is to help security practitioners decide whether this particular threat is worth the effort of implementing the defense. Attack Trees [97] can help focus where to spend a security budget, as well as keep track of which components depend on security of others, so that new attacks propagate throughout all their dependencies.

### 5.2.3 IoT Devices

A domain which symbolizes the rapid change in technology is the "Internet of things" (IoT). IoT may refer to any *thing* with an Internet connection, but especially devices which interact with the physical world, and which have only recently become connected to the Internet, such as refrigerators and light bulbs. Security of IoT devices is often an afterthought, and as a result, many compromised IoT devices have become part of botnets and used as part of DDoS attacks on secondary targets.

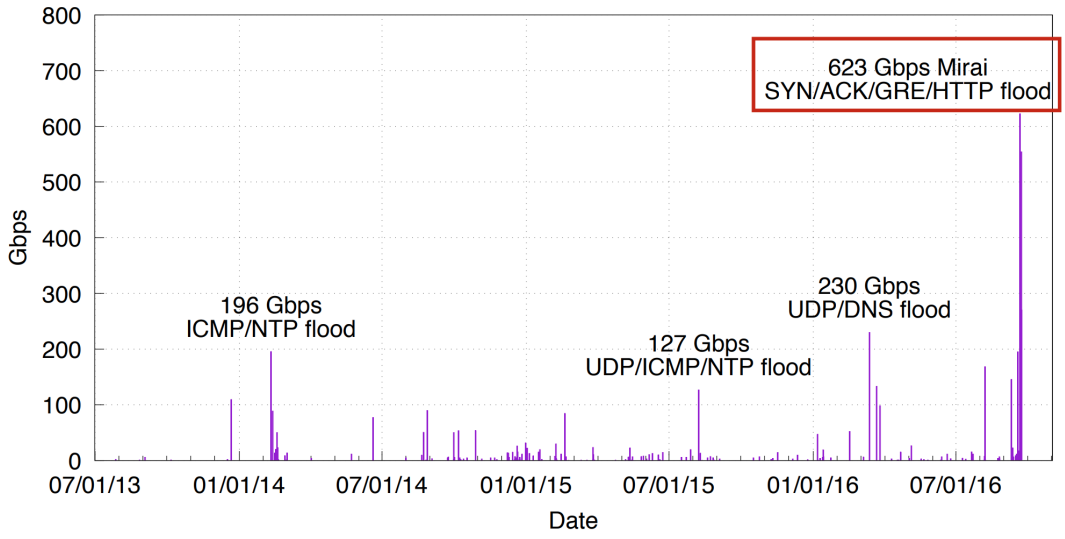



Figure 5.4: Mirai Botnet DDoS Attack [98]

One such botnet is Mirai, which as shown in Figure 5.4 in 2017 achieved the largest DDoS attack to date [98]. These attacks are difficult to defend against, because they closely resemble the activity of real users. Botnet scopes can be large, as illustrated by the Carna botnet in Figure 5.5, with 420 thousand active clients globally in a 9-month period. The IoT extends to "things" such as automobiles, where the consequences of attacks are more serious.

### 5.2.4 Physical Security

In the physical world, we have physical security. Walls keep our environments controlled, and keep threats out. Locks allow entry for those with keys. However, physical security is only considered for this document if it also pertains to information security. For example, when those locks connect to the Internet, the potential exists to unlock the door not with the relevant key, but by exploiting a software vulnerability.

### 5.2.5 Supply Chain

In the modern world, most goods and services are realized by combining a huge assortment of other goods and services. For example, a simple office chair may contain parts sourced from

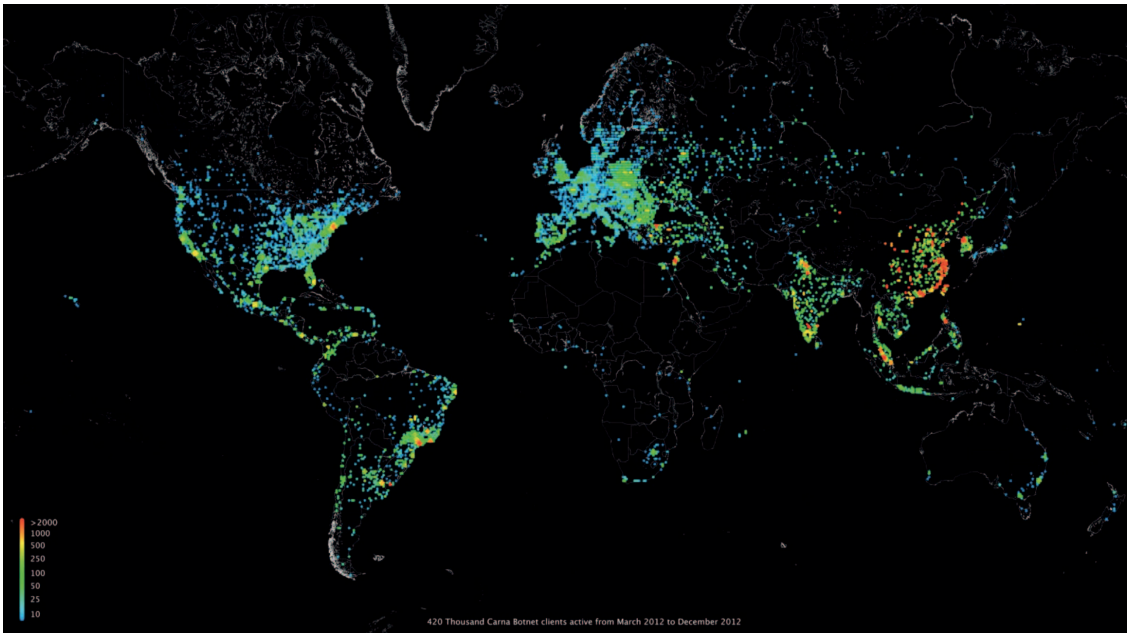



Figure 5.5: Carna Botnet DDoS Attack [99]

dozens of different countries, and may have been partially assembled in different countries. The flow of goods through this complex network of separate stages is called a supply chain. The supply chain has physical properties, such as where different parts are made or assembled. There are also logical aspects, such as who makes a particular part, and when. To manage this complexity, the modern supply chain is highly dependent on software. This software manages everything from logistical schedules to computer programs that operate industrial machines.

### 5.2.6 Software "Toolchains"

At the lowest level of the supply chain, below the hardware and software which powers the supply chain, is yet more software. This is the domain of compilers (programs that generate other programs) and hardware synthesis (programs that design hardware). These programs assist humans in creating the hardware and software artifacts which eventually power the entire supply chain and manage its complexity.

Popular modern compilers include Microsoft's MSVC, the Gnu Project's GCC, and LLVM's Clang compiler. With the exception of MSVC, these projects have an "open source" model, where anyone

can view the source code, and nearly anyone can propose modifications to the code, which are accepted with a kind of peer review process.

The open nature of these projects can be a liability or an asset, depending on your point of view. However, the economics of this open contribution model make it difficult for a single attacker to subvert the entire project, or at the very least makes such attacks harder to hide. This was not always the case. In the early days of compiler software, most development was done by researchers, and they rarely worked together on the same compiler. In the 1980s-1990s, proprietary commercial closed-source compilers were the most widely-used type of compiler. While these commercial compilers had some advantages, such as dedicated support for particular input languages and target architectures, they were often incompatible with each other, and would sometimes interpret the specifications and requirements of their input languages differently. Since the early 2000s, open source compiler projects have seen increased support, both in the form of contributions from volunteers, as well as paid sponsorship from corporate entities. Rather than invest in in-house compilers, many companies have seen value in contributing to popular open source compiler projects. This allows companies to still have some say in compiler development, but reduce their total engineering cost by not having to build the entire project alone.

## 5.3 Challenges

Technical challenges of logistics and economics dominate traditional supply chains, and cultural factors complicate information security. Today, most hardware synthesis is performed using proprietary closed-source tools provided by the same vendor that manufacturers the hardware itself. The situation is generally better for software, in large part due to the proliferation of high-quality free and open-source compilers such as gcc and clang. However, third-party software is increasingly commonly used in software to improve productivity. As can be seen from Figure 5.6, the hardware and software supply chain is very complex. Managing this complexity is a key challenge.

Another factor which slows adoption of security solutions is inertia. Individual people resist change because they are more comfortable with what they already know. Institutions resist change because

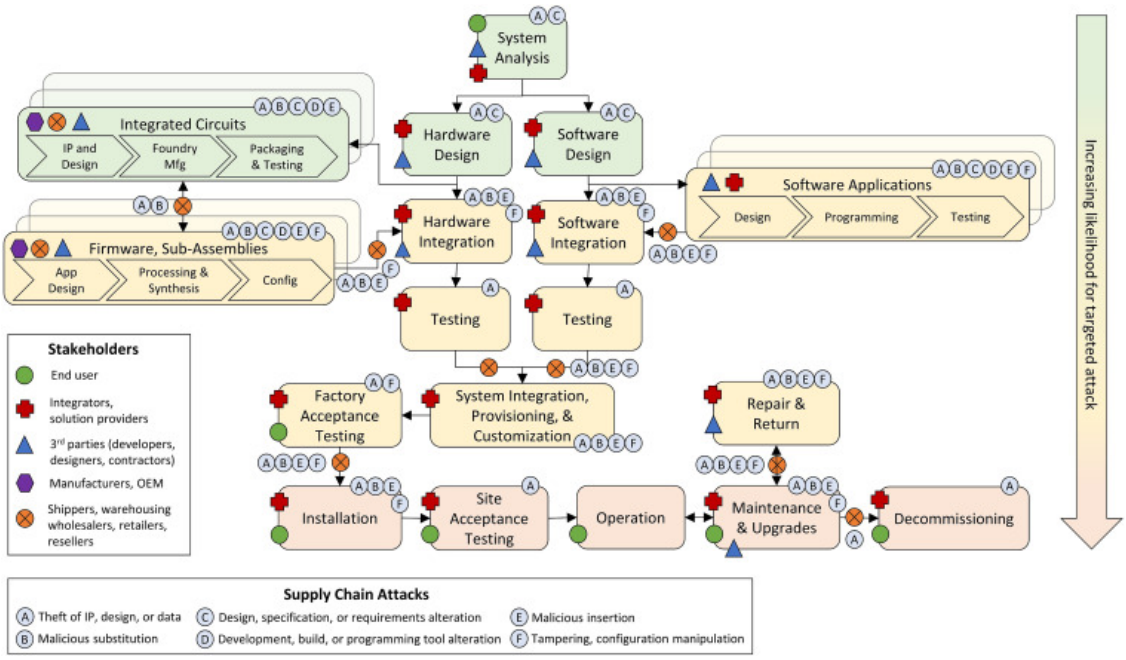



Figure 5.6: Hardware and Software Supply-Chain Complexity [95]

it requires a large investment of effort to change their processes - effort which could otherwise be spent improving products or seeking more customers. Even when a high-risk vulnerability is known, organizations may still sometimes fail [100] to implement timely fixes.

Political and organizational policies define what is important for many people. For example, in a startup the highest priority may be time-to-market, and security is a lesser concern. In governments, security may only become a priority if politicians sense sufficient political will from their constituents to support it.

## 5.4 Opportunities

Despite the complexity and challenges in the field of toolchains and information security, there are opportunities to improve best practices and the state of these arts. The first opportunity is to improve quality (of results as well as of user experience) of the toolchains. There are multiple methods to analyze the quality of software and hardware. This literature review will highlight some of these methods, such as static analysis using machine learning, and dynamic analysis using fuzz testing and symbolic execution.

Depending on how you measure, either computer hardware or computer software is the lowest "level" of our modern systems. Computer hardware ultimately executes all software. However, modern computer hardware is *designed* using a class of software called Electronic Design Automation (EDA) tools. If the EDA tools are compromised, then the hardware itself is compromised. As Ken Thompson [79] noted, it's more important to trust the people behind the tools, than the tools themselves.

Proving the presence of counterfeit or pirated hardware or software is currently a difficult task. This is due to many optimizations which can make even the same source code appear different after compilation or synthesis. If detection of counterfeit or pirated code was easier, companies would have an easier time prosecuting whoever illegally copied their code. Users of software would also be protected, because they could know with more certainty that they are using legitimate products. Most importantly, improving the best practices for securing our supply chains, from the lowest levels, strengthens the entire supply chain. While the effort to improve compilers and EDA tools may be significant, the benefits are astronomical, and investments in prevention can be more efficient overall than responding to individual incidents. Just as vulnerabilities propagate upward through all affected components, the benefit of each vulnerability fixed at the lowest level is multiplied by all the software or hardware which is built upon this level. Therefore, any improvements to compilers and synthesis propagate to all the applications of these tools.

## 5.5 Software Vulnerabilities and Mitigations

Finding vulnerabilities in software usually involves both static and dynamic analysis. Static analysis is faster, but typically can not find bugs which result from particular system states. Static analysis can include simple linear matching or more complex graph analysis. Linear (or "string") matching is less computationally intensive compared to graph techniques, but also less robust to minor code changes.

Dynamic analysis can find any run-time bug, in theory. However, in practice, it is too time-consuming to execute every possible state. To address this limitation, heuristic approaches, such

as fuzz testing, are used to test a subset of possible states.

While it is possible to statically analyze source code, this work focuses on analysis of binary files. Binary analysis has to contend with multiple challenges.

### 5.5.1 Deobfuscation

One challenge of binary analysis is that of obfuscation, where the meaning of the program is intentionally hidden. Udupa et al. describe multiple types of obfuscation. First is *surface* obfuscation, in which variable names are changed. Next is *deep* obfuscation, which changes control flow or data reference behavior [37]. For real-world binary analysis, surface obfuscation does not greatly inhibit program understanding, so we are mainly concerned with deep obfuscation.

When viewed as a graph, a typical program fragment such as the function in Figure 5.7 consists of multiple branches and loops. A type of obfuscation called *flattening* converts a program's Control Flow Graph (CFG) to a jump table structure, which transforms the shape of the graph. While flattening can make a program much more difficult to analyze, Udupa et al. show that flattening can be de-obfuscated statically. Their approach uses a technique called *constant propagation*, which is also typically part of compiler optimizations.

```
int f(int i, int j) {
    int a = 1;
    if (i < j)
        a = j;
    else
        do {a *= i--;}
        while (i > 0);
    return a;
}
```

Figure 5.7: Source Code [37]

The function's CFG and its flattened version are shown in Figure 5.8. Flattening dramatically changes the linear representation of the program, and hinders linear search based techniques. Binaries which were obfuscated with both flattening and intra-procedural control flow could not be de-obfuscated with a purely static approach. However, with dynamic analysis, functions could be de-obfuscated. A limitation of this dynamic approach is that it only works on functions which are actually called at run-time.

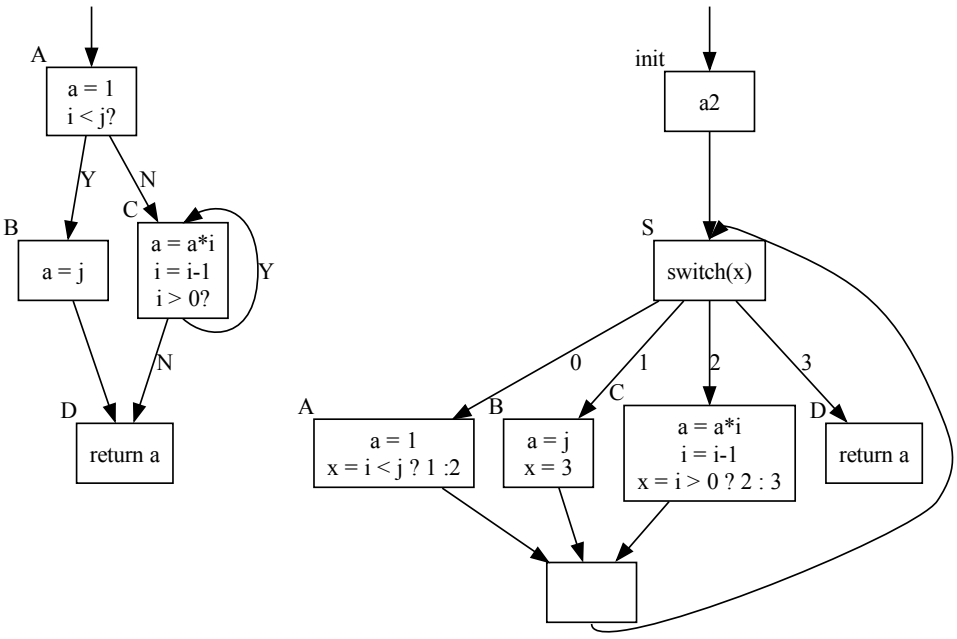



Figure 5.8: Left: Control Flow. Right: Flattened [37]

### 5.5.2 Concolic Testing

A technique called *concolic execution* or *concolic testing* combines symbolic execution with concrete values. The reason to combine these two types of analysis is that fuzzing (testing concrete values) is efficient to test, but untargeted. Meanwhile, symbolic execution can precisely find "interesting" values, but is much more computationally intensive to run. Symbolic testing is slow due

to the potential for combinatorial explosion of states, and is limited by the precision of the theorem prover or constraint solver used.

By combining fuzzing and symbolic execution, the fuzzer explores interesting code paths, and then the symbolic execution engine can "drill down" those paths. This saves time since the symbolic execution does not have to exhaustively search the entire state space. Stephens et al. [74] enhance this idea with "compartments" which categorize values which a program must take in order to reach a certain path, versus other valid or in-between values. Fuzzing mitigates the state space explosion problem, and uses a heuristic which counts state transitions to determine the interesting paths, rather than keep track of each state individually. The authors combine these features in a tool called *Driller*, emphasizing the "drill down" aspect of analysis.

### 5.5.3 Bug Signatures

By creating "bug signatures" in the intermediate representation IR, Pewny et al. are able to emulate I/O behavior to determine semantics [30]. They achieve this by sampling and hashing the sampled results. Their IR covers 3 distinct instruction set architectures (ISAs): x86, ARM, and MIPS. Even among firmwares with different ISAs, their sampling and hashing method is able to find vulnerabilities like Heartbleed, which was previously thought [101], [102] to be prohibitively difficult for signature-based methods to detect. Additionally, the method of [30] was able to find backdoors in closed-source MIPS and ARM-based router firmware.

### 5.5.4 Binary Analysis Techniques

Shoshitaishvili et al. [29] explain that there is a need to analyze binary files, and not just source code, because there can be malicious code embedded within the compiler, or which modifies code during transmission which can alter the actual code that executes. In addition to intentional malicious code, all steps of building and transmitting code can have errors. The authors produced a binary analysis engine, called *angr*, to try and consolidate efforts of many researchers in this area. Angr provides a framework for static analysis, dynamic analysis, and other features. For dynamic

analysis, angr facilitates concolic (concrete and symbolic) testing as well as fuzzing, as mentioned by [74].

The techniques described in [29] include static analysis, dynamic analysis, their analysis engine (angr), and others. They claim that static analysis is too "pessimistic", meaning it produces too many false positives when searching for problematic code. Despite that, angr still benefits from some forms of static analysis. For example, control flow graph (CFG) recovery requires analyzing multiple different categories of indirect jumps:

- computed
- context-sensitive (as in the case of callback functions)
- object-sensitive (as in the case of virtual functions, especially when analyzing code which came from compiled C++ sources)

Graph-based vulnerability discovery is another aim of this static analysis. The aim here is to search for examples of already-known vulnerable within the given code. By maintaining a database of already-discovered vulnerabilities, both in machine code and source code, this type of search can be extremely fast. Shoshitaishvili cites [30] as a source of this method. The downside to this type of matching-based search is that copied or malicious code often is obfuscated to hide its meaning from exactly this type of detection.

The dynamic analysis techniques used by angr include concolic testing and fuzzing.

The analysis engine, *angr*, uses an intermediate representation (IR) taken from the Valgrind project called *libVEX*, and handles multiple different binary formats. The engine also handles symbol resolution and relocation. Their state representation component, *SimuVEX*, handles register values, symbolic and abstract memory, as well as commonly used libraries like POSIX. This interfaces with plugins like *Solver* and *Architecture*.

The additional features afforded by *angr* include crash replay, and it can be used for tasks like exploit hardening as well. Overall, the goal of *angr* is to be a framework for facilitating multiple different forms of analysis.

## 5.6 Software Compilation

Generally, compilation transforms one language to another. Usually, the transformation is from a "high level" language to a relatively "low level" language. For example, translating C++ source code into x86-64 executable machine code. Unless otherwise mentioned, the type of compilation is usually "ahead of time" or AOT. There are other types such as "just in time" or JIT compilation, which does some of the compilation at run-time. This is subtly different from an "interpreter", which interprets source code at run time and translates it into executable code, although things like JIT compilation blur this distinction. The reasons for compiling a language usually include: better run time performance, more extensive static analysis to catch bugs prior to run time, and an ahead-of-time compiler can perform extensive optimizations which would be too computationally costly to perform at run time Sometimes, developers choose to compile their programs because it can obscure the ideas of the source code. This obfuscation is due to compilation being a "lossy" process. Some information such as comments and variable names may be removed by the compiler, which makes it harder for a human reader to understand the intent of the code.

### 5.6.1 Compiler Construction

Traditionally, because the input language and output language are both syntactically and semantically different, the process of compilation is split into multiple distinct parts. Often, compilers are described in terms of having a *front end*, which recognizes one or more languages, and produces an intermediate representation as its output. As shown in Figure 5.9, this intermediate representation may be further transformed by multiple other phases in the *middle* section of a compiler, including various forms of optimization. Ultimately, the results of the middle phases are transformed by a *back end* which produces machine code for a target architecture. As the front end may accept more than one source code language, so the back end may emit code for more than one architecture.
A common way to categorize all these phases in the compiler pipeline with finer granularity than *front*, *middle*, and *back*, is given by [103]. These finer-grained phases are: tokenizing, lexical

analysis, syntax analysis, context handling, and code generation. Any of these steps may be further decomposed into smaller tasks.

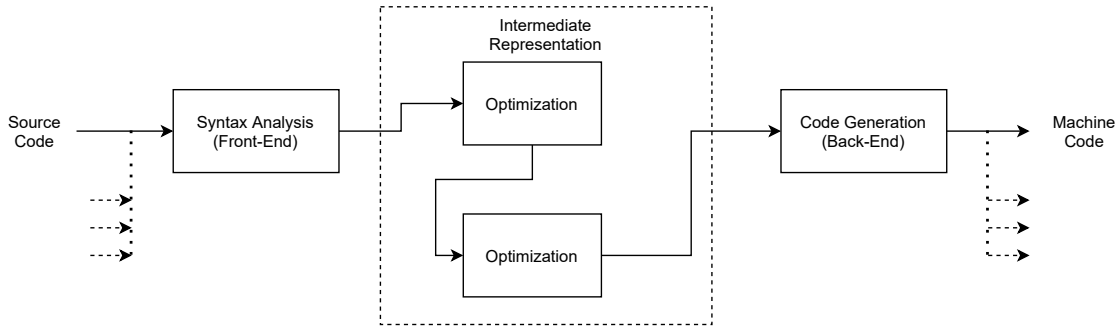



Figure 5.9: Compiler Phases

Tokenizing usually implies reading the characters or bytes of a text representation of a program into larger, more meaningful pieces, or *tokens*. For instance, separating words from punctuation and whitespace characters. Lexical analysis is a further refinement of this task, in which usually some form of longest-string matching finds particular words and classifies them according to the language specification. For example, the C language has the keyword called “while”, which has a particular meaning to the language. But C does not have a keyword called “meanwhile” So even though the keyword “while” is a substring of “meanwhile”, a correct tokenizer for the C language would not treat “meanwhile” as containing the keyword “while”. Lexical analysis is strictly an analysis of regular language, and as such can be processed with regular expressions. Syntax analysis handles aspects which can not be handled by regular languages, such as contextual information. This aspect of compilation can be used to distinguish between tokens based on where they appear in the program. For example C has a keyword “while” but it also has strings of characters. If the characters ’w’ ’h’ ’i’ ’l’ ’e’ appear in sequence within a string, this should not be treated as the keyword, but rather as part of the string. Usually syntax analysis results in a representation of the program as an abstract syntax tree (AST) which encodes more structure of the program. Together, lexical analysis and syntax analysis are often referred to as “parsing”. Context handling augments the AST form with environmental or contextual information, for instance to help determine the types of variables. Finally, code generation produces a result in the form of the

output language of the compiler.

### 5.6.2 Nanopass Compiler Framework

This word describes a framework [104] for writing compilers, which emphasizes the creation of multiple intermediate languages tailored to make particular transformations easier. Nanopass basically uses a separate IR for each transform. This splits the overall compilation task into many separate steps. For example, type checking can be one "pass" out of many. Each pass produces an intermediate language, which may have no relation to the input or output languages.

### 5.6.3 LLVM and Intermediate Languages

It is worth noting that in modern compiler projects such as the Gnu Compiler Collection (GCC), multiple choices of input and output languages are supported. But because of this separation of concerns, handling multiple different input languages can be achieved by adding a different set of "front ends", one for each language, which deal with the concrete syntax of the input language.
The Low-Level Virtual Machine (LLVM) [31] is a modular compiler framework, which aims to consolidate effort from many unaffiliated compiler researchers into a single framework.
For example, the world's top register allocation experts can work on LLVM's register allocator module, alongside parsing experts working on its parser module. Compiler writers can then "plug-in" modules from these separate parts for their own specific input-output language requirements.
Internally, LLVM uses a form called Static Single Assignment, in which each variable is assigned to exactly once. This form permits some optimizations which are not possible if a variable's value is allowed to change throughout the run time of the program. LLVM also uses a simple language-independent type system, and has language-independent exception handling facilities (for example the setjmp/longjmp library in C which provides exceptions to the usual programmatic control flow). This intermediate representation allows different modules to inter-operate, and for compiler writers to come up with novel combinations. For example obfuscation can be a module, and it can be combined with a separate optimization module.

### 5.6.4 Intermediate Representations for Different Systems

An extension to the ideas of LLVM, Multi-Level IR or MLIR [105] aims to provide a similar framework and modular system, but to handle the case of compilers for very different targets, from GPU/CPU/TPU differences, all the way to the difference between single-core CPU and multiple-machine distributed system. By treating these differences with the same theoretical framework, MLIR authors hope to achieve the same benefits of LLVM's modular architecture, but for vastly different target architectures.

### 5.6.5 OpenCL

OpenCL [106] is an open language specification aimed at efficient computation using heterogeneous processing elements. The basic architecture consists of a single host working with one or more compute devices, each with one or more compute elements. Its execution model uses the idea of *kernels* which execute on the compute elements and are managed by the host program.

OpenCL's memory model supports several scopes (global, workgroup-local, and private) as well as immutable (constant) global memory. Allocation strategies are dynamic for the host and static for kernels.

OpenCL supports both *data parallel* and *task parallel* execution models. Hierarchical parallelism can be managed explicitly by the programmer or implicitly by the OpenCL implementation.

Since OpenCL aims to offer a single language interface for orchestrating work across heterogeneous platforms, this simplifies the task of finding vulnerabilities which leverage flaws at different application layers. Also, by having a formalized memory, execution, and programming model, testing can target both host and kernel endpoints simultaneously and use the same interfaces. A unified programming language and model also reduces the amount of duplicated code and provides fewer opportunities for bugs and vulnerabilities to appear in the first place. Finally, if FPGA designers create hardware which adheres to the specifications of the OpenCL model, then testing applications which are built on that hardware can be simplified.

### 5.6.6 Compiler Vulnerabilities

Compilers are the "lowest level" of the software supply chain. Nearly everything is built on top of the results of compilers, from terminals to text editors to web browsers. If one of these components is not built directly by a compiler, it is likely that it is built by a separate component which in turn was built by a compiler. Compilers are used to build the software which is in turn used to design hardware, so in a sense, compilers are "below" the level of modern computer hardware. Because of this fundamental nature of compilers, a compiler has nearly unlimited power. And a malicious compiler therefore is extremely dangerous to society.

In [79], Thompson illustrates that it is possible to insert malicious code into a compiler, then compile the compiler's source code, and remove the traces of the malicious code from the source, leaving behind the malicious executable code. Thompson ensured that all the tools built by this compiler would cooperate with the malicious code, such that even attempting to disassemble the malicious compiler binary would reveal no evidence of the attack. Further, Thompson designed the compiler so that it would propagate its malicious code when used to compile a source code that was *not* malicious.

The motivation for an attacker to perform this type of attack is extremely high, due to the compiler being used to generate so many other parts of a system. Detecting this type of attack is also extremely difficult, because the malicious code takes steps to hide itself from an investigator.

Fortunately, Wheeler found a counter to this attack in [80]. The basic idea to countering this attack relies on a separate compiler and careful composition and comparison of separate compiler results. Start with a "trusted compiler" binary $T$, and unknown compiler binary $A$. Use $A$ to compile the source code for $A$, testing whether $A$ is capable of self-regenerating, and obtain the result binary $B$. The fact that $A$ can self-regenerate does not indicate that it contains malicious code, but self-regeneration is necessary to replicate the Thompson attack. Next, compile $A$'s source using $T$, resulting in a new binary $T_{A1}$. Then, compile $A$'s source using $T_{A1}$, resulting in a new binary $T_{A2}$. Finally, compare $A$, $T_{A1}$, and $T_{A2}$ in a trusted environment. To be trusted, the environment should be at minimum free of utilities which were compiled by $A$. If all three of $A$, $T_{A1}$, and $T_{A2}$

are bitwise identical, then the source of $A$ "accurately reflects" $A$. If not, then $A$ may contain the Thompson attack.

There are some caveats to this mitigation. First, $T$ has to implement any non-standard extensions in the same way as $A$. All the compilers in question must be deterministic. And most importantly, $T$ must be trustworthy.

This may mean building $T$ yourself so you can be confident that does not contain the malicious code. However, in practice, it is considered reasonable to simply obtain a *different* compiler for $T$, because the chance of two separate compilers containing exactly the same malicious code is perhaps small. The two compilers $A$ and $T$ can *both* actually be malicious, but as long as they are not malicious in the same way, this method will detect an inequality between $A$, $T_{A1}$, and $T_{A2}$.

As Balakrishnan et al. point out, "What You See Is Not What You Execute" [107]. For any given piece of software, the source code is the part intended for humans to read. Source code is the interface between the programming system and the human designers, so it is designed for humans to express their intent, and (hopefully) easy for other humans to read. Binary code, by contrast, is designed for the computer to execute and offers no provisions for human readability. Therefore, most people do not read the binary code, but instead read the source code of a program. However, the source code is what the computer actually executes. Any vulnerabilities, errors, or malicious code must therefore exist in the binary form. But because of the difficulty in reading and interpreting binary code, finding vulnerabilities in binary code is non-trivial. Further compounding this difficulty is the fact that errors in compilers can introduce vulnerabilities in binary code, even when the source code contains no error. One extremely common source of this type of error is "undefined behavior" in compilers. This is a situation in which the language does not specify the semantics of some string of syntactically valid code. In these situations, a compiler can do anything at all while still claiming to faithfully compile. Compiler writers may choose to cause the compilation to fail if undefined behavior is detected. However, in other cases, the compiler may proceed to compile the program, and make some arbitrary choice of what to do with the code which triggers the undefined behavior.

## 5.7 Firmware

"Firmware" generally refers to the program which manages initial device configuration, or which is intended to run with full control over the device's hardware. In systems which do not have operating systems, the firmware is often the only program that runs on the device. In systems with operating systems, the firmware typically runs at initial power on time, to configure the hardware in preparation for giving control to the operating system. Internet of Things (IoT) devices and embedded systems (such as automotive computer systems) are classes of systems in which the firmware is significant. As previously mentioned, fuzz testing or fuzzing is a dynamic analysis method to explore code paths in a running system. Feng et al. [89] note the difficulties of applying fuzz testing to firmware and propose a methodology to work around these difficulties.

The main challenge in fuzzing firmware is interacting with peripherals. Peripherals are hardware devices attached to the computing device, so testing normally requires exercising the physical device hardware and observing its real-world effects. Even if these peripherals can be virtualized, the input/output (I/O) interfaces are still usually the slowest part, and their implementation as a physical interface makes parallelism more difficult to implement correctly. Feng et al. note that I/O can be 3 orders of magnitude slower than "native" (e.g. CPU-memory) operations. Further challenges to fuzzing firmware include limited microcontroller (MCU) emulator support and the fact that many firmware systems use custom operating systems, real-time operating systems, or "bare metal" libraries.

The approach by [89] called Processor-Peripheral Interface Modeling (P2IM) solves these challenges and permits the use of generic fuzz testing software to test firmwares. First, no physical or emulated peripherals are used. Instead, they use a concept called "approximate MCU emulation" in which the exact responses of a peripheral are not required, but instead use the much more relaxed requirement that the responses from a peripheral do not cause the system to crash. P2IM also provides a method to generate such approximate MCU emulators, and to create abstract template MCUs, for example based on the ARM Cortex-M MCU. This process infers firmware-specific in-

formation via a process they call “explorative firmware executions”, and this technique allows for model instantiation on-demand.

## 5.8 Control Flow Graph (CFG) Analysis

### 5.8.1 CFG Algorithm Comparison

In their paper, Chan et al. show a method to detect similarity in software [17]. One of the co-authors of this paper is the primary Tigress developer, a software obfuscation framework. They note that real-world CFGs tend to have some specific properties, such as out-degrees of 2 or less (unless the CFG includes a switch statement or an exception). They also often resemble series-parallel graphs, and are usually reducible to small basic blocks of between $\approx 4$ to 7 instructions.

While general-purpose graph similarity measures are expensive to compute, it is possible to leverage heuristics about real-world CFGs to improve analysis time. Their method uses predefined CFGs with known edit distances to evaluate CFG similarity. Graph edit distance is defined, along with a cost function for the different edit operations such as add/delete node, or add/delete edge.

This method ranks graph similarity algorithms according to this cost function, by first generating CFGs of increasing edit distance, then running the algorithm under test and ranking its results. It compares the current algorithm’s rank to the ground truth using a **sortedness** or **Pearson correlation** to obtain a score of quality for the algorithm. A limitation of this study is that it only studies graph *topology*, so if it encounters two different sets of instructions that nonetheless have the same *topology*, then they would be considered equal.

In this study, four algorithms are compared:

- Kruegel, which uses a fingerprinting technique
- Hu, which is based on edit distance
- Avujosevi, which uses neighbor matching, and
- Sokolsky, which is simulation-based.

Overall, they find that the Hu algorithm obtains the highest score.

### 5.8.2 discovRE

An approach to cross-architecture bug finding called *discovRE* uses signal processing techniques, such as k-nearest neighbors (KNN) and graph analysis techniques such as maximum common subgraph isomorphism (MCS) [22]. The authors note that compiler optimization makes binary bug search an NP-complete problem. To reduce the search space, discovRE pre-filters its inputs, and focuses on a "bug database" similar to the approach of [30]. Their approach extracts features such as:

- number of instructions
- size of local vars
- number of basic blocks
- function CFG

Then uses a numeric filter (KNN) and a structural filter (MCS) on the disassembled binary to find the bugs based on their similarity to those in its database. To improve the results, discovRE adds diversity to their database by using 4 different compilers, covering both x86-based and ARM-based architectures, and obtains source code from multiple open source software projects.

### 5.8.3 Zynamics Bindiff

A method for detecting similarities in binary files, with an emphasis on graph matching, is called Bindiff and integrated into the IDA Pro reverse engineering software as an extension [21]. This extension also includes some features to match known standard library functions, but a limitation is that it can only match functions for which it has already acquired a fingerprint.

## 5.9 Machine Learning for Software Reverse Engineering

Advancements in GPUs and neural network theory in recent years have revitalized the field of machine learning. Early successes in computer vision and natural language processing have inspired others to try to use machine learning (ML) for reverse engineering. More recently, large repositories of source code have been used as training data for AI assisted code generation. However, inferring meaning from code has proven more challenging than generating code from "docstring" text [108].

A common challenge of applying machine learning to any task is availability of data. Even a well-funded machine learning company recently canceled its successful robotics lab due to lack of data [109]. Therefore, the barrier to applying machine learning to software is lower than to hardware, primarily because generating compiled software is faster.

### 5.9.1 CNN and Data Augmentation

Applying an already developed approach, such as a convolutional neural network (CNN) can produce good results if the data is a good fit for the model. Catak et al. create an "image" from a malware using term frequency - inverse document frequency (TF-IDF) representation [7]. They then apply data augmentation using additive noise, similar to computer vision applications which apply noise to real images, in order to artificially increase the sample data size. The authors claim that adding Poisson noise to this image representation achieves perfect accuracy.

### 5.9.2 Siamese Networks for Similarity

Liu et al. [28] note that previous approaches [20]–[23] all rely on CFGs and "expert knowledge", which may introduce bias. Existing solutions which consider binary semantics [19], [20] rely on computationally expensive theorem proving, which is sensitive to minor changes in semantics due to patch changes. This sensitivity gives too many false negative results, essentially classifying a binary as "different" when it should instead classify it as a "near match". Instead, as noted by

[110], it is possible to train a neural network on the raw bytes of the compiled binary and achieve good results without disassembly or preprocessing.

Liu et al. describe a network (using pairs of match- and non-match functions) with CNN as input which embeds the matrix-encoded input as a feature vector prior to ingestion by a Siamese network with Euclidean loss function. This basic architecture is shown in Figure 5.10.

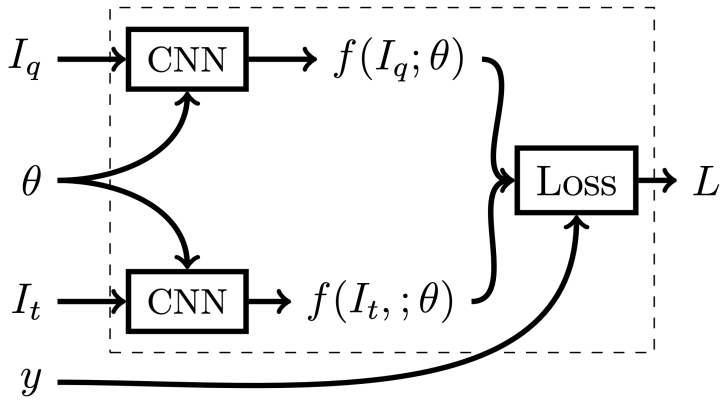



Figure 5.10: CNN $\rightarrow$ Siamese Model

### 5.9.3 Asm2Vec

One approach called Asm2Vec by Ding et al. [13], tries to categorize and detect clones in binary software. Clone detection is difficult for traditional methods due to both software optimization and intentional obfuscation. Existing SOTA approaches at the time of this paper do not capture relationships between features and assume features are independent. For example, LSH-S [27], ngram [111], n-perm [111], BinClone [112], and Kam1n0 [113] use frequency counts for operators and operands in a disassembled binary. Meanwhile Tracelet [114] uses edit distance, and discovRE [22] and Genius [23] use synthesized features such as arithmetic instruction to control transfer instruction ratios, or basic block counts. Asm2Vec uses obfuscator-llvm [26], an obfuscation module for the LLVM compiler toolchain, to produce diverse binaries from known inputs that use one or a combination of obfuscation modes:

- Bogus Control Flow Graph

- Control Flow Flattening
- Instruction Substitution

Ding et al. attempted to use Tigress to obfuscate the source code itself, rather than the IR, but could not get the generated code to correctly compile. Asm2Vec learns a 200-dimensional vector for assembly functions, and is able to classify functions into logically correct categories. For example, when viewed in a 2-dimensional PCA projection, functions such as `memcpy` and `strcpy` are close to each other, and `file` related functions are also grouped together.

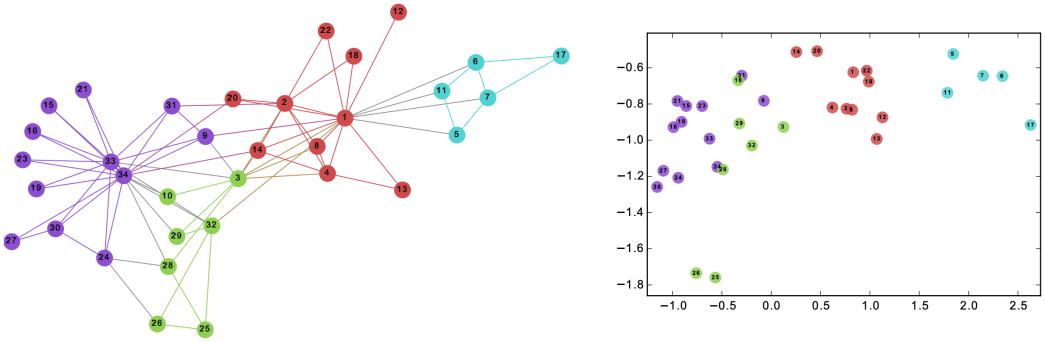
### 5.9.4 Malware Detection from Small Samples

Hasegawa et al. show that even a simple one-dimensional CNN can reliably discriminate between malware and "goodware" [115]. Even more impressive is achieving 97% accuracy from only the last 1024 bytes of the binary file.

### 5.9.5 Graph Embedding

Rather than rely on manual feature engineering, Massarelli et al. use a recurrent neural network (RNN) to extract features automatically [116]. They try to show compiler provenance by binary similarity measurements, and compare their approach of unsupervised feature learning to manual feature engineering. This approach uses graph embedding to achieve improved results over manual feature engineering.

Graph embedding itself is described by Xu et al. in [58]. They describe a method which they call Structure2Vec, which uses a Siamese network to determine similarity. Their default policy is that "same functions are similar" and "non-same functions are different". Their model uses cosine similarity to determine similar or different scores for a given pair of functions. To evaluate their results, they compare against codebook graph embedding of Genius [23] which contains 33,045 firmware images. Xu et al. claim the neural network does a better job of learning features than graph matching algorithms.



(a) Input: Karate Graph (b) Output: Representation

Figure 5.11: DeepWalk Latent Representation [49]

As shown in Figure 5.11, DeepWalk [49] illustrates that a model can learn a latent representation of graph data in a lower-dimensional space. The intuition of this approach is that random walks through a graph are a close approximation of semantic connections between nodes in a graph. This approach is used to classify text data.

Similarly, Node2Vec also uses a random walk procedure to sample neighborhoods, which allows to "smoothly interpolate between BFS and DFS". However, this random walk procedure is *biased* with edge weights, so choosing the next node can be influenced by these weights as shown in Figure 5.12.

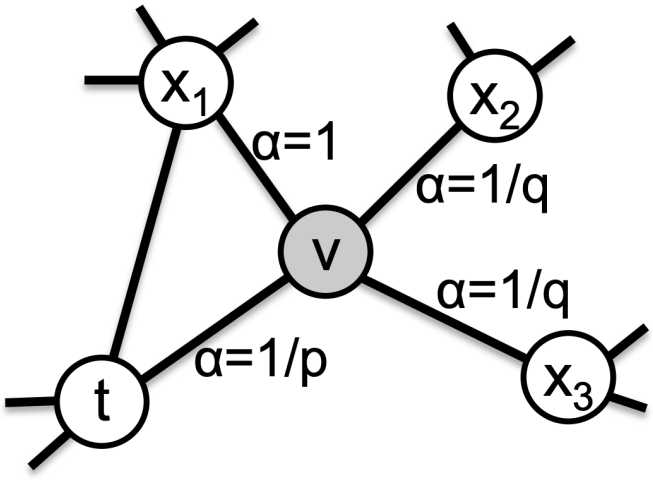



Figure 5.12: Random Walk [6]

### 5.9.6 Language-Independent Algorithm Detection

Another project which uses a Siamese network [5], by Nghi et al., obtained $\approx$ 3500 implementations of 6 different algorithms from GitHub. These algorithms include classic homework problems such as *merge sort*, *bubble sort*, *knapsack*. The languages used in the implementations include Java and C++. The goal is to detect if two separate programs, which use different programming languages, implement the same algorithm or not.

### 5.9.7 Deep Networks

Gupta et al. use a multi-layer network with an attention mechanism [12] on a dataset of student homework submissions with errors. The goal of this model is to fix coding errors automatically. Their results are promising: 27% of errors can be completely fixed, and 19% can be partially fixed, all without manual intervention.

### 5.9.8 Attention for Graph Neural Networks

Thekumparampil et al. note that a linear neural network model without fully connected layers performs well on graph data [117]. This approach reduces the total number of required parameters, and by adding an *attention* mechanism, weights among neighbors can show how these neighbors influence each other. Allamanis et al. noted that it is possible to use an attention mechanism for summarizing source code in their work, but this did not consider obfuscated source code nor compiled binary artifacts [118].

### 5.9.9 Language-Independent Code Semantic Learning

Ben-Nun et al. define a novel IR that describes contextual flow, and create a graph embedding called "inst2vec" [119]. Their model uses a recurrent neural network (RNN) on inst2vec graph embeddings and the authors claim SOTA performance.

## 5.10 Symbolic Execution

Symbolic Execution is a powerful technique which is somewhat limited by its computational expense. Shoshitaishvili et al. describe a system called Firmalice [75], which tries to find backdoors or authentication bypass in software binaries. Firmalice was later integrated into *angr*. Another tool, called Miasm lifts assembly code into an IR, in order to use symbolic execution and run the IR with specific values and emulate a code's semantics [120]. It is used by the Sibyl project to "identify functions from their side effects".

In an effort to improve the efficiency of symbolic execution, Wang et al. reduced part of the constraint solving task into a Markov decision process [121]. Their experiment shows an improvement over existing heuristic-based approaches. They used this to develop a greedy algorithm to approximate the optimal result. This approach depends on a sub-function which finds a local optimal strategy, such as a solver or concrete execution. Another project, called PANDA, uses the QEMU emulator and a compact representation to find "repeatable traces" of a program [122]. Similarly, Egele et al.'s approach [36] executes functions in a sandbox and analyzes their side effects. This approach leverages graph isomorphism and heuristics to improve analysis time.

## 5.11 Graph Algorithms

Modern machine learning techniques are fundamentally statistical techniques which operate in a continuous state space. Therefore, being able to model graphical data relationships, which are inherently discrete, in continuous space, is an important first step to applying machine learning to linked, relational, or generally graph-shaped data. Some fundamental graph techniques include searching or traversing graph data structures, either in Breadth-First Search (BFS) or Depth-First Search (DFS) order. These search patterns have different properties in terms of memory usage, run time, and ability to find global optima. In addition to search, there are other graph techniques applicable to EDA security, some of which are summarized here.

### 5.11.1 Graph Isomorphism

As noted in many of these works, both programs and hardware descriptions may naturally be represented as graphs. Finding similarities between graphs (graph isomorphisms) is an important part of analyzing hardware and software binary code. Subgraph isomorphism is an NP hard problem [123], and the first practical algorithm appears in 1976 by Ullmann. Ullmann's algorithm [124] "inferentially eliminates successor nodes" for efficiency, and uses an adjacency matrix to find where the subgraph similarity stops. Finally, it prunes the subgraph to avoid searching where no similarity is possible.

In software, many projects are built using standard libraries of functions, so that programmers can focus on their unique application, and take advantage of general-purpose system functionality developed by others. Because of the common reliance on standard library functions, reverse engineers can benefit from some method of automatically detecting them within binaries. At the time of Qiu et al.'s paper on this subject [44] in 2015, the approach for identifying library functions in binary code is the FLIRT plugin for IDA Pro [38]. FLIRT uses pattern matching on the first 32 bytes of a disassembled function, and is confused by both inlining and compiler instruction reordering. Qiu et al. achieve two results in their paper:

- they extend the CFG with an Execution Dependence Graph
- they find library functions by subgraph isomorphism

Finding library functions in stripped binaries is naturally more difficult than finding them in binaries which retain all their symbols. An approach by Jacobson et al. compares favorably to FLIRT even though it also uses a library fingerprinting technique [125]. While FLIRT's approach is not robust to compiler optimization differences, the semantics are the same across all variations. By only fingerprinting the wrapper functions for *exported* glibc functions, they claim to achieve 1.67 times better accuracy than FLIRT.

## 5.12 Human Factors

Ultimately, RE is performed for a reason, and the human in the loop decides what is important and what to search for. Humans doing any task are limited by the capacity of their working memory (WM). This limits things like the number of variables a person can keep track of at a given time, and exceeding this capacity causes WM errors to occur. There are multiple different types of WM errors, and they depend on the person, the program tracing strategy (such as top-down or bottom-up) and the program style (such as imperative or functional). Chrichton et al. recommend [126] some ways in which to reduce WM errors:

- reduce variable scope, so variables are used near their definitions
- show variables to programmers
- externalize program state
- provide writable display interfaces for code where readers can add annotations

The authors note that some common practices, such as function parameters, may violate these recommendations (by effectively causing the variable's state to be lexicographically distant from where it is used). However, they also point out that some programming environments, such as DrRacket and the Lean language mode within Emacs, can assist with showing variables to programmers and help externalize the program's state.

## 5.13 Simulation vs. Formal Verification [127]

It is relatively easy to generate tests to be run in simulation. However, obtaining useful test coverage is much more difficult. For example, pseudo-random simulation is easy to specify, but produces incremental tests of low value. High-coverage testing is also resource-intensive. To illustrate this, [127] notes that test simulation farms, consisting of hundreds of computers, may run for weeks in order to obtain even moderate test coverage, but full coverage by this method is still intractable due

to state space explosion. Emulators, constructed from FPGAs, may sometimes be used to speed up simulation through hardware acceleration, but constructing them costs significant engineering effort, and so this method is reserved for only designs which expect to sell in high volume. Meanwhile, formal verification can provide the equivalent of 100% coverage, by proving that a system meets each of its specifications. However, implementing the formal verification test is more difficult, and in practice only done for simple modules.

### 5.13.1 Model-Based Formal Verification

Model-based verification is attractive because it can be fully automated. Essentially, this type of verification performs a brute force exploration of the solution space. However, these models are limited in depth due to state explosion and can only model some hundreds of latches. Symbolic simulation tests multiple inputs in parallel by propagating a symbolic function vector through the design.

### 5.13.2 Proof-Theoretic Methods

This approach uses theorem proving software to show that the model matches its specification. This approach is used in conjunction with hierarchical design and abstractions to keep the scope of each part manageable. However, this approach requires extensive human guidance, making it impractical to scale to large designs.

### 5.13.3 Symbolic State Traversal

This method uses breadth-first search, which is linear in the number of variables and exponential in number of memory elements. Because of this exponential factor, computing the "image" of states reachable from the current state can take too long. If a bug is found by this approach, constructing the input trace which produced the bug is also difficult.

### 5.13.4 Symbolic Simulation

By propagating symbolic expressions, instead of concrete values, through a simulated circuit, a full boolean description of the circuit is obtained. This allows testing in parallel as well as comparing to the boolean expression of the formal design.

This can be implemented as an iterative algorithm which runs for N steps, and at each step computes the outputs of the boolean circuit and, if the circuit has sequential logic feedback, saves them as inputs for the next iteration step.

The approach in [128] uses an "abstract" approach to find non-reachable states, compacting the search space. Their observation is that the complex boolean expressions at each simulation step have more information than is required to identify reachable states, so a more compact (or "compressed") encoding can be used to classify reachable versus non-reachable states. This compressed encoding allows better simulation scalability, and permits exploration into the range of a few thousand latches, an order-of-magnitude improvement over the uncompressed approach.

### 5.13.5 Summary of [127]

These approaches improve the efficiency of logic simulations, which is essential for determining if the logic design is correct and matches specifications. However, they do not consider the interaction of the design with higher level protocols and applications. Testing at any level higher than logic gate simulation is not considered, and so bugs at any higher level are out of scope for [127]. It is at this higher level where side-channel vulnerabilities like Spectre and Meltdown occur.

Specifically, the techniques described by [127] do not address side-channel attacks. Therefore, there is a clear need for testing methods which consider application-level defects whose root cause manifests at the hardware level. Otherwise, despite using formal verification techniques, automatic test generation, and concrete or symbolic simulation methods, side channel vulnerabilities will be a persistent problem.

## 5.14 OpenCL [106]

OpenCL is an open language specification aimed at efficient computation using heterogeneous processing elements. The basic architecture consists of a single host working with one or more compute devices, each with one or more compute elements. Its execution model uses the idea of *kernels* which execute on the compute elements and are managed by the host program.

OpenCL's memory model supports several scopes (global, workgroup-local, and private) as well as immutable (constant) global memory. Allocation strategies are dynamic for the host and static for kernels.

OpenCL supports both *data parallel* and *task parallel* execution models. Hierarchical parallelism can be managed explicitly by the programmer or implicitly by the OpenCL implementation.

### 5.14.1 How OpenCL Relates to Hardware/Software EDA Security

Since OpenCL aims to offer a single language interface for orchestrating work across heterogeneous platforms, this simplifies the task of finding vulnerabilities which leverage flaws at different application layers. Also, by having a formalized memory, execution, and programming model, testing can target both host and kernel endpoints simultaneously and use the same interfaces. A unified programming language and model also reduces the amount of duplicated code and provides fewer opportunities for bugs and vulnerabilities to appear in the first place. Finally, if FPGA designers create hardware which adheres to the specifications of the OpenCL model, then testing applications which are built on that hardware can be simplified.

## 5.15 High Level Synthesis

In FPGA synthesis tools such as Xilinx's Vivado, High Level Synthesis (HLS) permits writing code with C-like syntax and semantics, and then deploying that code as a hardware configuration on an FPGA. For nearly all varieties of HLS in use today [129], including Xilinx's own HLS, the C-level code is first compiled to synthesizable RTL code, which then goes through the standard FPGA

synthesis toolchain steps such as technology mapping, place and route, and bitstream generation [130]. Besides Vivado HLS, other types of HLS include [131]–[133] and many others as mentioned by [134]:

> ...Among these tools are Mentor Catapult C Synthesis, Forte Cynthesizer, Celoxica Agility compiler (sold in 2008 to Catalytic, which renamed itself Agility Design Solutions), Bluespec, Synfora PICO Express and Extreme, ChipVision PowerOpt, NEC CyberWorkBench, AutoESL AutoPilot, Xilinx AccelDSP (which started as the product of an independent company, AccelChip) and SystemGenerator, Esterel EDA Technologies Esterel Studio, Synopsys Synplicity Synplify DSP, and, just announced in the summer of 2008, Cadence C-to-Silicon (C2S) compiler.

With rare exceptions, every one of these projects compiles a C-like input language (typically C or C++) into Verilog or VHDL as its output language. RTL in general, and Verilog specifically, is therefore an intermediate representation within the overall compilation process of transforming HLS code into a hardware configuration. One possible reason why so many HLS implementations output Verilog is that the synthesis tools do not expose any other input format.

### 5.16 Alternatives to HLS

Proprietary software like Vivado exposes some interfaces to scripting the tool, with varying levels of granularity. Both current and previous versions of Xilinx's FPGA synthesis tools, and Altera's Quartus EDA tool, provide interfaces to their placement and routing engines, to allow external tools to customize these phases to varying degrees. Modern FPGA synthesis tools also provide internal scripting, usually by a Tool Command Language (TCL) interpreter with access to selected commands.

TCL's main drawback is its slow execution speed, and more recent tools like RapidWright have achieved nearly 2 orders of magnitude speedup (88x faster) in the task of importing look-up tables (LUTs) into Vivado by bypassing the TCL interpreter [135]. Nevertheless, scripting the synthesis

tool remains a popular method of allowing user-defined extensions to the scripting tool.

## 5.17 Graph Representation in EDA Tools

### 5.17.1 In FPGA Synthesis

According to [136], netlists are commonly represented as graphs:

> Existing CAD tools represent their netlists with graph structures. The range of internal representations of these graphs is not too large - they usually consist of an array-of-structs, where each net has forward and backward pointers to each net it drives/is driven by.
>
> In general this solution works well but requires the entire design to be flattened which expands the memory footprint and limits the tool's ability to exploit multiply-instantiated modules for performance.

Using graph algorithms, FIRRTL performs dead code elimination, constant propagation (through modules), determines a module's clock domain and clock crossings, as well as detecting combinational logic loops, or asserting that all paths between two signals take exactly N cycles [136]. However, the optimizations done by FIRRTL are separate from those done by downstream tools. According to Brayton et al, logic synthesis uses environmental information, such as signal arrival and timing requirements, parasitics, and don't-care conditions, to produce a "correct implementation which meets timing and testability constraints and minimizes area". Logic synthesis primarily involves combinational logic, and sequential logic is treated separately, at least at the time of publication of [137] (1990) [137]. When describing the background for their machine learning-based approach to logic optimization, Neto et al. mention [138] that two primary representations are used for optimizing logic: either an And-Inverter Graph (AIG) or a Majority-Inverter Graph (MIG). A MIG output is true when a majority of its three inputs are true, and both permit negation of edges, so both graph data structures are able to describe universal logic. Boolean circuits are partitioned

into subgraphs and the major contribution of [138] is to automatically choose either MIG or AIG representation to achieve more optimal results without human intervention.
Several papers [139]–[141] describe how logic minimization and other aspects of CAD/EDA workflows can be formulated as graph problems. In particular, graph algorithms that correspond to EDA tasks include the following list (compiled by [140]):

- **set covering** for logic partitioning [142], [143]
- **state encoding** [139], [144]–[146]
- **logic minimization** [147]
- **planar routing** [148]

### 5.17.2 In Compilers

Compilers commonly express many parts of a program as graph data structures, and use graph algorithms on these data structures [149]. For example, expressing the dependencies of variables and values from earlier to later statements results in a directed, acyclic dependency graph. Transformations on this graph data structure include renaming, expansion, node splitting, and forward substitution [149]. Graph structures are also used to represent data flow and control flow [150], and this facilitates modifications such as searching, merging, inserting, and deleting nodes or edges from the graph in the course of various optimizations, such as minimizing the length of critical paths and minimizing the number of array access nodes in the flow graph. In their Essentials of Compilation, Siek et al. describe [151] a compiler architecture as primarily consisting of transformations on abstract syntax trees (ASTs), a type of graph data structure. Several important operations within the compiler framework they describe are formulated as graph algorithms, such as

- building an interference graph for liveness analysis
- implementing graph coloring for register allocation
- copying a graph as one part of garbage collection (Cheney's algorithm [152])

- mapping expressions to control flow graphs to support conditional expressions
- converting ASTs into control flow graphs

Interprocedural dependence analysis is performed by [153] using a "value flow" graph (VFG) after converting the linked program to static single assignment (SSA) form, which itself is computed using graph algorithms and stored as a graph data structure [154]. Graph algorithms are heavily used in compilers as noted by [155], in relation to the following tasks associated with compilation:

- depth first walks to find strongly connected components
- constructing intervals to facilitate flow graph reduction
- reachability analysis (for constructing def-use chains)
- liveness analysis
- finding "busy" expressions as candidates for code hoisting
- constant folding
- finding dominators

### 5.17.3 In Both

Recently, there is renewed interest in the heuristic-based Espresso [156] algorithm, long used in FPGA and ASIC EDA tools for logic minimization, and itself an improvement over the early Quine-McCluskey algorithm [156]. Kanakia et al. recognized that within deep learning network inference, large and sparse (many "don't care" states) boolean functions are often generated, and so they devised a GPU-based parallel improvement over Espresso-II (97x faster) [157].

# CHAPTER 6

# DISCUSSION

## 6.1 Results and interpretation

The most important contributions of this dissertation are the following three contributions:

1. a method of data augmentation based on source code obfuscation which facilitates large-scale binary synthesis from few source samples

2. multiple methods of processing compiled binary data into formats which can work with existing deep learning models

3. an example application which detects known functions within unknown binary code

Some of our findings are in line with what we expected to find based on our survey of related literature. Our fundamental assumption held true: training on larger and more diverse datasets improves the quality of a machine learning model's results. A key challenge of this work was determining which methods of synthesizing larger datasets are appropriate for binary compiled program data, and evaluating those with both existing and new deep learning models. Regarding the utility of binary code-specific specialized feature engineering versus generalized or data-agnostic model approaches, in this work we converged on a balance which uses some of both. In particular, the use of disassemblers proved to be critically valuable in the series of steps used to transform binary data into a format usable by deep learning models. With few exceptions, deep learning models require their inputs to be in the form of vectors of numbers. Representing the input as a bag-of-words at the byte level proved more effective than using a bag-of-words at the disassembly text token level. Still, investing some effort to transform binary program data into vectors of numbers made it possible to evaluate existing machine learning models and more quickly explore the design space of this domain. We also noted that considering the graph structure of binary or assembly code improved model quality, although this result is not as strong as expected. However, the utility of this

graph representation is not limited to its use as an input format, because the ability to synthesize random walks through an existing control flow graph affords an additional form of obfuscation. In our experiments we included a fixed number of random walks, each of a fixed length. However, the number and length of these random walks is another hyperparameter which can be varied at will both during the dataset augmentation phase as well as during the training phase.

Some related works use methods for data augmentation inspired from different fields such as natural language processing (NLP) and computer vision, with varying degrees of success. In some cases, binary data is interpreted as flattened image data and reshaped from a linear sequence into a rectangular matrix to be used with existing computer vision models [11]. These approaches attain some successes, but we argue that such reshaping is not appropriate for binary data, because computer vision techniques such as two-dimensional convolutions and two-dimensional pooling inherently create spatial meaning: nearby image pixels are often related to each other, but binary data arbitrarily reshaped into rectangles has no such semantics.

The fields of computer vision and NLP can use random noise effectively for data augmentation, but we found this to be detrimental for binary program classification. One possible explanation is that noise in natural language data may take the form of spelling errors or insertion, deletion, or transposition of words within a document. Such noise often does not drastically alter the meaning of the larger sentence or document. In computer vision, added noise may imply shifting the color values of random pixels within an image. Unless a large fraction of the pixels are replaced with noise, we can usually still recognize the original image. However for binary program data, the addition of a small amount of noise does not change the meaning of the program by a correspondingly small amount. Rather, a single wrong bit often results in an *invalid* program. In short, what works for augmenting images or text does not necessarily work for augmenting programs.

Of the related works that employ obfuscation, such as [26] and [37], most use obfuscator-llvm or custom machine code transformations to achieve obfuscation of the binary code. By contrast, we use Tigress which applies transformations at the source code level, transforming C source code inputs into obfuscated C source code outputs. Because the outputs share the same format as the in-

put, it is natural to apply further transformations to the already-transformed outputs until a desired level of complexity is reached. We show that machine learning models trained on obfuscated code are more robust to new obfuscation types, making this approach a good choice for analyzing code which could have been obfuscated by an adversary. We use an existing machine learning model which is already effective for analyzing code and improve its performance by adding obfuscation to its training dataset.

There are multiple ways to account for the graph structure of code. Some approaches, such as [69], extract the control flow graph and then represent it as an adjacency matrix in order to analyze it with a two-dimensional convolutional neural network (CNN). Others like [13] and [66] create embedding vectors based on an instruction and its context. Most similar works which utilize some form of the code's control flow graph (CFG) obtain this graph using the reverse engineering software IDA Pro. In our work, we extract CFGs using angr, with settings that result in disassembly similar to that produced by IDA Pro. Whereas IDA is proprietary closed-source software with a commercial license, angr is open-source with a permissive license. This makes our approach more accessible to researchers with limited funds.

In multiple additional models we show that the representation of the input code matters and offers trade-offs between training time, inference time, and quality of result.

## 6.2 Limitations

This work relies on a high quality source-to-source obfuscating tool such as Tigress, and inherits its limitations. For example, Tigress only works with C source code, and that code must be self-contained in a single file and it must include a "main" function. However, while C is prolific and used in critical software infrastructure across many domains, it is not the only language or execution environment which could benefit from better static analysis or improved security. The obfuscator-llvm project integrates with the llvm compiler tools to add obfuscation to any language which can be compiled by llvm. While the work described in this dissertation does not use obfuscator-llvm, we note other related works which do. Today, the primary reason for us to prefer

Tigress over obfuscator-llvm is that Tigress permits more varied and higher absolute amounts of obfuscation due to the fact that its output is in the same format as its input. This enables us to iteratively layer additional transformations one on top of the other, resulting in much more diversity of output.

Obfuscator-llvm only has a handful of obfuscation types and to the best of our knowledge they can not easily be combined sequentially to arbitrarily increase the obfuscation amount in the compiled artifact. More importantly, these transformations occur within the intermediate representation (IR) used internally by llvm, limiting the iterative combination of multiple transformations to those that can operate on this IR. Because llvm supports multiple languages in addition to C, obfuscator-llvm inherits that capability. We believe that despite these transformations happening in the IR level, the presence of multi-language support will likely make obfuscator-llvm a preferable choice in the future as languages other than C become more popular for low-level systems tasks. Long term, it seems more practical to add IR-to-IR transformations to obfuscator-llvm than to build separate source-to-source obfuscators for every language llvm already supports.

In this work we consider only one instruction set architecture (ISA) and focus primarily on one operating system (GNU/Linux). The broader world of software is much more diverse. Supporting additional ISAs will broaden the scope of this work, but a related area of research already focuses on cross-architecture similarity. These related works attempt to find the commonalities between similar code which has been created for different ISAs and/or different operating systems. Implicit in these works is the assumption that there exists a useful abstract representation of code which can be synthesized from the artifacts compiled for different environments. However, we note that there already exists such an abstraction in the form of the original source code. But compilation is a lossy process so these cross-architecture similarity methods do not aim to decompile into legible source code, but identify major commonalities.

Dataset bias is inherent in our approach because we are essentially transforming a small number of training samples into a larger number of samples. Data augmentation in general is likely to amplify existing bias in a dataset, and our approach is no different in this regard. To mitigate this

bias to some extent, we ensure that the number of samples for each training class are approximately balanced. This prevents any single class from dominating the training set. However, we see some evidence of this form of bias in the function prediction results in Figures 4.7 and 4.8, particularly noticeable in the way our model overestimates the likelihood that the test program contains certain functions, such as the `daemon` function.

The use of standard library functions biases our dataset toward compact and single-purpose functions. We determined this to be an acceptable trade-off based on their relevance to reverse engineering more broadly [38] as well as their familiarity to a general audience.

This dissertation utilizes graph features in different experiments - first in the ensemble model of Chapter 3, and in a different way in the Block-Walk embedding model of Chapter 4. The first method is only concerned with extracting graph metadata as another feature dimension for the model. The second method incorporates graph features via random walks, which imparts some elements of the program's possible runtime behavior into a static representation that is safe to analyze even if the program contains malware. A notable limitation of both of these methods of incorporating binary program data's graph structure is that neither is used in the embedding phase. Recent works such as [158] note that self-supervised learning in graph neural networks, particularly when trained on multiple tasks, can outperform supervised methods. Similarly, [159] achieve high performance on node clustering using a self-supervised approach. Compilers generate predictable and idiomatic graphs of basic blocks, but certain subgraphs are likely to be more valuable at identifying different functionality. Even in the presence of obfuscation, which may inject confounding block subgraphs, there should be some idioms that persist and help identify the function. Therefore, self-supervised learning may improve results in our task at multiple phases: both during embedding the data and during classification.

In this work we described the types of obfuscating transformations applied, but we spent much less time describing the obfuscations we did *not* use. One type of obfuscation in particular is relevant because it is highly effective: virtualization. In this type of obfuscation, rather than implementing the code directly, the obfuscated program describes a virtual machine and the behavior is realized

by actions taken by that virtual machine. This obfuscation method is more difficult to reverse engineer than simpler transformations that encode literals or modify flow control. This work avoided analyzing this kind of transformation for a mundane reason: Tigress support for virtualization is error-prone. This might be a relief for the security analyst or reverse engineer; if it is hard to automatically generate virtualized versions of a program using Tigress, then malware authors are less likely to use it. But this relief will not last indefinitely, because Tigress is not the only obfuscation tool, will likely improve in the future, and virtualizing functions has always been possible to implement manually.

### 6.3 Unexpected results

In our experiment using a PV-DM model inspired by [13], we find a large improvement by using an ensemble of voting classifiers, but no additional improvement by considering the graph metadata of the input programs. This is contrary to our expectation that including metadata about the program's graph structure should enhance the model's capability. On the other hand, a subsequent experiment using the Block-Walk embedding model shows an improvement when including random walk data. One possible explanation for this apparent disagreement between results is that the graph metadata is not in a format as suitable for its model as the random walk data. In the PV-DM model, the inputs are tokens while its graph metadata portion uses a Random Forest classifier. It is possible that the number of features is too small for this type of classifier. Conversely, the random walk data is in the same format as the basic blocks, and integrates naturally with the existing model.

Another unexpected result is the surprising effectiveness of the deflate+kNN model, and its performance in classifying stripped data. We expected such a model to perform better when analyzing an input program that contains debug symbols, but instead the impact on accuracy is negligible. In addition, because the stripped programs are smaller, this model is able to produce results faster.

Finally, while we expected our Random Forest model to perform better when given disassembly tokens as input, what we find instead is that modeling the input bytes as a bag of words is more effective.

## 6.4 Future research

One enhancement to this research is to include a broader set of functions within the training set. Starting with a full set of standard library functions is one way to increase the breadth of the dataset, but even better would be to include functions from third-party libraries or application code. Another incremental but practical enhancement to this work is to include a mapping between the predicted function and its position in the input binary. This would permit integration with existing tools such as IDA Pro, Ghidra, or angr to pinpoint exactly those regions of the binary that are of interest to the analyst.

One of our original research goals was to apply this approach to domains outside of source code, such as to electronic design automation (EDA) tools that produce configurations for FPGAs or ASICs. However, based on the amount of training data we believe to be necessary, existing Verilog or VHDL based code generators are likely too slow, and to the best of our knowledge there is no freely available repository of register transfer level (RTL) code suitable for training. Adapting the methods of this dissertation to the EDA domain would therefore require finding a suitable data representation which is amenable to an analogous method of data augmentation.

Lastly, this dissertation does not explore a transformer-based approach, although based on recent successes of transformer-based architectures in language modeling including [118], it seems likely that a similar architecture would also work well for binary program data. One gap we identified in the related work is that some models use a pre-trained BERT model [70] but this model is pre-trained on natural language data rather than on assembly or binary code data. Based on our results we believe it would be valuable to experiment with a BERT model which has been pre-trained on assembly or basic block tokens rather than natural language. Such a model would embody a kind of language understanding for binary code which is fundamentally different from the language understanding of natural languages, and possibly be better suited to tasks related to binary analysis.

# CHAPTER 7

# CONCLUSIONS

## 7.1 Structure

Binary code has a unique structure compared to natural language data, images, or audio data. From the available methods for data augmentation which could be applied to code, we find obfuscation is most suitable. Certain types of data, such as images or text, may be altered in subtle ways with corresponding slight differences in the meaning of the transformed result compared to the original. But in the case of binary compiled code, subtle changes typically result in invalid data. This makes traditional methods of data augmentation poorly suited to binary analysis. Using a source-to-source obfuscator facilitates arbitrarily high amounts of obfuscation and more diverse training data.

## 7.2 Representation

In addition to choosing the right strategy for data augmentation, it is also critical to choose the right representation of this data. By extracting basic blocks of a program's control flow graph, we can treat each block as a token within a vocabulary of blocks. This is more effective than representing a program as a bag of words at the individual byte level, or using assembly language tokens, or even separating instruction opcodes from their operands. We find that training a model for a classification task rather than predicting a token within its context also results in higher performance for classifying regions of code according to the functions the model saw in training. More importantly, treating basic blocks as the fundamental token affords treating random walks and sequences of basic blocks in a uniform way with a single deep learning model. This flexibility makes it trivial to attach different weights to basic blocks versus walks through the same data. While domain-agnostic approaches can analyze binary data, their effectiveness is limited. In this work we find that deep learning based methods which take into account the unique structure of code are consis-

tently better than domain-agnostic approaches. By treating basic blocks as words in a vocabulary, we are able to strike a good balance of performance and accuracy. Most promising is the use of models which, when trained on a mixture of plain and obfuscated data, are able to generalize from both data categories. Such models enable better insights for reverse engineers without requiring massive databases of sample code, and more importantly this approach is resilient to the challenges of reality in which attackers have asymmetric advantage over defenders.

### 7.3 Application

Finally, this work demonstrates the first implementation of a general purpose method for identifying known code within unknown, obfuscated, and stripped binary data. The scope of this initial work is limited to a single ISA and a handful of small functions, but the underlying principles can readily be adapted to different training data and different ISAs without loss of generality. Practical applications of this work should consider training on first-party code for tasks such as intellectual property identification, regression testing, or performance analysis. Conversely, as malware sample databases grow, security researchers can train on malware data to better detect new variants before they are able to cause damage. Even before malware samples become available, this work makes it easier and more practical to automate the process of finding regions of code within an unknown program which are most interesting to a security analyst or reverse engineer. We believe this dissertation demonstrates a viable and practical path forward for a new family of deep learning based static analysis and is fertile ground for continued research.

## REFERENCES


[1] R. K. Vaidya, L. De Carli, D. Davidson, and V. Rastogi, "Security issues in language-based sofware ecosystems," *Arxiv preprint arxiv:1903.02613*, 2019.

[2] D.-L. Vu, I. Pashchenko, F. Massacci, H. Plate, and A. Sabetta, "Typosquatting and combosquatting attacks on the python ecosystem," in *2020 ieee european symposium on security and privacy workshops (euros&pw)*, 2020, pp. 509–514.

[3] R. Duan, O. Alrawi, R. P. Kasturi, R. Elder, B. Saltaformaggio, and W. Lee, "Towards measuring supply chain attacks on package managers for interpreted languages," *Arxiv preprint arxiv:2002.01139*, 2020.

[4] D. Hendler, S. Kels, and A. Rubin, "Detecting malicious powershell commands using deep neural networks," in *Proceedings of the 2018 on asia conference on computer and communications security*, 2018, pp. 187–197.

[5] N. D. Q. Bui, L. Jiang, and Y. Yu, "Cross-language learning for program classification using bilateral tree-based convolutional neural networks," *Corr*, vol. abs/1710.06159, 2017, Available: `http://arxiv.org/abs/1710.06159`

[6] A. Grover and J. Leskovec, "Node2vec: Scalable feature learning for networks," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 855–864.

[7] F. O. Catak, J. Ahmed, K. Sahinbas, and Z. H. Khand, "Data augmentation based malware detection using convolutional neural networks," *Peerj computer science*, vol. 7, p. e346, 2021.

[8] A. K. Biswas, "Cryptographic software ip protection without compromising performance or timing side-channel leakage," *Acm transactions on architecture and code optimization (taco)*, vol. 18, no. 2, pp. 1–20, 2021.

[9] C. Taylor and C. Collberg, "Getting revenge: A system for analyzing reverse engineering behavior."

[10] M. Bayer, M.-A. Kaufhold, B. Buchhold, M. Keller, J. Dallmeyer, and C. Reuter, "Data augmentation in natural language processing: a novel text generation approach for long and short text classifiers," *International journal of machine learning and cybernetics*, pp. 1–16, 2022.

[11] N. Marastoni, R. Giacobazzi, and M. Dalla Preda, "A deep learning approach to program similarity," in *Proceedings of the 1st international workshop on machine learning and software engineering in symbiosis*, 2018, pp. 26–35.

[12] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Proceedings of the aaai conference on artificial intelligence*, 2017, vol. 31.

[13] S. H. Ding, B. C. Fung, and P. Charland, "Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *2019 ieee symposium on security and privacy (sp)*, 2019, pp. 472–489.

[14] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space." arXiv, 2013. doi: 10.48550/ARXIV.1301.3781.

[15] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *International conference on machine learning*, 2014, pp. 1188–1196.

[16] "Musl." https://musl.libc.org.

[17] P. P. Chan and C. Collberg, "A method to evaluate cfg comparison algorithms," in *2014 14th international conference on quality software*, 2014, pp. 95–104.

[18] NSA, "Ghidra." https://ghidra-sre.org/.

[19] Y. David, N. Partush, and E. Yahav, "Statistical similarity of binaries," *Acm sigplan notices*, vol. 51, no. 6, pp. 266–280, 2016.

[20] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, "Bingo: Cross-architecture cross-os binary search," in *Proceedings of the 2016 24th acm sigsoft international symposium on foundations of software engineering*, 2016, pp. 678–689.

[21] H. Flake, "Structural comparison of executable objects," 2004.

[22] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "Discovre: Efficient cross-architecture identification of bugs in binary code.," in *Ndss*, 2016, vol. 52, pp. 58–79.

[23] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 acm sigsac conference on computer and communications security*, 2016, pp. 480–491.

[24] W.-C. Chao, "Asm2vec-pytorch." https://github.com/oalieno/asm2vec-pytorch.

[25] "Radare2." https://www.radare.org/n/index.html.

[26] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-llvm–software protection for the masses," in *2015 ieee/acm 1st international workshop on software protection*, 2015, pp. 3–9.

[27] A. Saebjornsen, *Detecting fine-grained similarity in binaries*. University of California, Davis, 2014.

[28] B. Liu *et al.*, "Alphadiff: cross-version binary code similarity detection with dnn," in *Proceedings of the 33rd acm/ieee international conference on automated software engineering*, 2018, pp. 667–678.

[29] Y. Shoshitaishvili *et al.*, "Sok: (state of) the art of war: Offensive techniques in binary analysis," 2016.

[30] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *2015 ieee symposium on security and privacy*, 2015, pp. 709–724.

[31] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," in *International symposium on code generation and optimization, 2004. cgo 2004.*, 2004, pp. 75–86.

[32] S. Schrittwieser, S. Katzenbeisser, J. Kinder, G. Merzdovnik, and E. Weippl, "Protecting software through obfuscation: Can it keep pace with progress in code analysis?," *Acm computing surveys (csur)*, vol. 49, no. 1, pp. 1–37, 2016.

[33] R. Wagner, "Modern static analysis of obfuscated code," in *Proceedings of the 3rd acm workshop on software protection*, 2019, p. 1.

[34] J. Singh and J. Singh, "Challenge of malware analysis: malware obfuscation techniques," *International journal of information security science*, vol. 7, no. 3, pp. 100–110, 2018.

[35] M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns," 2003.

[36] M. Egele, M. Woo, P. Chapman, and D. Brumley, "Blanket execution: Dynamic similarity testing for program binaries and components," in *23rd usenix security symposium (usenix security 14)*, 2014, pp. 303–317.

[37] S. K. Udupa, S. K. Debray, and M. Madou, "Deobfuscation: Reverse engineering obfuscated code," in *12th working conference on reverse engineering (wcre'05)*, 2005, p. 10–pp.

[38] I. Guilfanov, "Ida fast library identification and recognition technology (flirt technology): In-depth," *[2012-03-11]. https://hex-rays.com/products/ida/tech/flirt/in_depth/*. 2012.

[39] J. McMaster, "Issues with flirt aware malware." 2011.

[40] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs," *Arxiv preprint arxiv:1808.04706*, 2018.

[41] M. Franz, "E unibus pluram: massive-scale software diversity as a defense mechanism," in *Proceedings of the 2010 new security paradigms workshop*, 2010, pp. 7–16.

[42] H. Dai, B. Dai, and L. Song, "Discriminative embeddings of latent variable models for structured data," in *International conference on machine learning*, 2016, pp. 2702–2711.

[43] S. Yu, T. Wang, and J. Wang, "Data augmentation by program transformation," *Journal of systems and software*, vol. 190, p. 111304, 2022.

[44] J. Qiu, X. Su, and P. Ma, "Library functions identification in binary code by using graph isomorphism testings," in *2015 ieee 22nd international conference on software analysis, evolution, and reengineering (saner)*, 2015, pp. 261–270.

[45] Q. Mi, Y. Xiao, Z. Cai, and X. Jia, "The effectiveness of data augmentation in code readability classification," *Information and software technology*, vol. 129, p. 106378, 2021.

[46] W. Jin, L. Zhao, S. Zhang, Y. Liu, J. Tang, and N. Shah, "Graph condensation for graph neural networks," *Arxiv preprint arxiv:2110.07580*, 2021.

[47] A. Loukas, "Graph reduction with spectral and cut guarantees," *Journal of machine learning research*, vol. 20, no. 116, pp. 1–42, 2019, Available: `http://jmlr.org/papers/v20/18-680.html`

[48] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," *Advances in neural information processing systems*, vol. 30, 2017.

[49] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," in *Proceedings of the 20th acm sigkdd international conference on knowledge discovery and data mining*, 2014, pp. 701–710. doi: 10.1145/2623330.2623732.

[50] C. Taylor and C. Colberg, "A tool for teaching reverse engineering," 2016.

[51] J.-M. Borello and L. Mé, "Code obfuscation techniques for metamorphic viruses," *Journal in computer virology*, vol. 4, no. 3, pp. 211–220, 2008.

[52] R. Tarjan, "Depth-first search and linear graph algorithms," *Siam journal on computing*, vol. 1, no. 2, pp. 146–160, 1972.

[53] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[54] A. Marcelli, M. Graziano, X. Ugarte-Pedrero, Y. Fratantonio, M. Mansouri, and D. Balzarotti, "How machine learning is solving the binary function similarity problem," in *31st usenix security symposium (usenix security 22)*, 2022, pp. 2099–2116.

[55] I. U. Haq and J. Caballero, "A survey of binary code similarity," *Acm computing surveys (csur)*, vol. 54, no. 3, pp. 1–38, 2021.

[56] M. von Tschirschnitz, "Library and function identification by optimized pattern matching on compressed databases: A close to perfect identification of known code snippets," in *Proceedings of the 2nd reversing and offensive-oriented trends symposium*, 2018, pp. 1–12.

[57] A. Di Federico, M. Payer, and G. Agosta, "Rev. ng: a unified binary analysis framework to recover cfgs and function boundaries," in *Proceedings of the 26th international conference on compiler construction*, 2017, pp. 131–141.

[58] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 acm sigsac conference on computer and communications security*, 2017, pp. 363–376.

[59] Y. David, U. Alon, and E. Yahav, "Neural reverse engineering of stripped binaries using augmented control flow graphs," *Proceedings of the acm on programming languages*, vol. 4, no. OOPSLA, pp. 1–28, 2020.

[60] S. Yang, L. Cheng, Y. Zeng, Z. Lang, H. Zhu, and Z. Shi, "Asteria: Deep learning-based ast-encoding for cross-platform binary code similarity detection," in *2021 51st annual ieee/ifip international conference on dependable systems and networks (dsn)*, 2021, pp. 224–236.

[61] L. Massarelli, G. A. Di Luna, F. Petroni, R. Baldoni, and L. Querzoni, "Safe: Self-attentive function embeddings for binary similarity," in *Detection of intrusions and malware, and vulnerability assessment: 16th international conference, dimva 2019, gothenburg, sweden, june 19–20, 2019, proceedings 16*, 2019, pp. 309–329.

[62] A. Vaswani *et al.*, "Attention is all you need," *Advances in neural information processing systems*, vol. 30, 2017.

[63] X. Li, Y. Qu, and H. Yin, "Palmtree: Learning an assembly language model for instruction embedding," in *Proceedings of the 2021 acm sigsac conference on computer and communications security*, 2021, pp. 3236–3251.

[64] F. Xia *et al.*, "Graph learning: A survey," *Ieee transactions on artificial intelligence*, vol. 2, no. 2, pp. 109–127, 2021.

[65] C. Eagle, *The ida pro book*. no starch press, 2011.

[66] X. Jin, K. Pei, J. Y. Won, and Z. Lin, "Symlm: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings," in *Proceedings of the 2022 acm sigsac conference on computer and communications security*, 2022, pp. 1631–1645.

[67] K. Pei *et al.*, "Symmetry-preserving program representations for learning code semantics." 2023.

[68] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith, "No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations.," 2015.

[69] Z. Yu, R. Cao, Q. Tang, S. Nie, J. Huang, and S. Wu, "Order matters: semantic-aware neural networks for binary code similarity detection," in *Proceedings of the aaai conference on artificial intelligence*, 2020, vol. 34, pp. 1145–1152.

[70] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *Arxiv preprint arxiv:1810.04805*, 2018.

[71] V. Andalibi, *Leveraging machine learning for end-user security and privacy protection*. Indiana University, 2022.

[72] A. Shroyer and D. M. Swany, "Data augmentation for code analysis," in *2022 international conference on intelligent data science technologies and applications (idsta)*, 2022, pp. 156–163.

[73] A. Qasem, M. Debbabi, B. Lebel, and M. Kassouf, "Binary function clone search in the presence of code obfuscation and optimization over multi-cpu architectures," in *Proceedings of the 2023 acm asia conference on computer and communications security*, 2023, pp. 443–456.

[74] N. Stephens *et al.*, "Driller: Augmenting fuzzing through selective symbolic execution," 2016.

[75] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, "Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware," 2015.

[76] Z. Jiang, M. Yang, M. Tsirlin, R. Tang, Y. Dai, and J. Lin, ""low-resource" text classification: A parameter-free classification method with compressors," in *Findings of the association for computational linguistics: Acl 2023*, Jul. 2023, pp. 6810–6828. doi: 10.18653/v1/2023.findings-acl.426.

[77] P. Bojanowski, E. Grave, A. Joulin, and T. Mikolov, "Enriching word vectors with subword information," *Arxiv preprint arxiv:1607.04606*, 2016.

[78] S. Y. A. Fayi, "What petya/notpetya ransomware is and what its remidiations are," in *Information technology-new generations: 15th international conference on information technology*, 2018, pp. 93–100.

[79] K. Thompson, "Reflections on trusting trust," in *Acm turing award lectures*, 2007, p. 1983.

[80] D. A. Wheeler, "Countering trusting trust through diverse double-compiling," in *21st annual computer security applications conference (acsac'05)*, 2005, p. 13–pp.

[81] W. Van Eck, "Electromagnetic radiation from video display units: An eavesdropping risk?," *Computers & security*, vol. 4, no. 4, pp. 269–286, 1985.

[82] A. R. Javed, T. Baker, M. Asim, M. Beg, and A. H. Al-Bayatti, "Alphalogger: Detecting motion-based side-channel attack using smartphone keystrokes," 2020.

[83] A. Moradi, D. Oswald, C. Paar, and P. Swierczynski, "Side-channel attacks on the bitstream encryption mechanism of altera stratix ii: facilitating black-box analysis using software reverse-engineering," in *Proceedings of the acm/sigda international symposium on field programmable gate arrays*, 2013, pp. 91–100.

[84] M. Seaborn and T. Dullien, "Exploiting the dram rowhammer bug to gain kernel privileges," *Black hat*, vol. 15, p. 71, 2015.

[85] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space aslr," in *2013 ieee symposium on security and privacy*, 2013, pp. 191–205.

[86] E. Dijkstra, "7.4 structured programming," *Software engineering techniques*, p. 65, 1970.

[87] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "Safe systems programming in rust: The promise and the challenge," *Communications of the acm*, 2020.

[88] T. Hoare, "Null references: The billion dollar mistake," *Presentation at qcon london*, vol. 298, p. 88, 2009.

[89] B. Feng, A. Mera, and L. Lu, "P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling," in *29th USENIX security symposium (USENIX security 20)*, Aug. 2020, pp. 1237–1254. Available: `https://www.usenix.org/conference/usenixsecurity20/presentation/feng`

[90] O. Analytica, "Us pipeline hack to make ransomware risks a priority," *Emerald expert briefings*, no. oxan-ga.

[91] N. Sganga, “Jbs paid $11 million ransom after cyber attack. \text{jhttps} : //www.cbsnews.com/news/jbs - ransom - 11 - million/, 2021.$

[92] C. Beek *et al.*, “Mcafee labs threat report,” https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-sept-2017.pdf, White Paper, 2017.

[93] A. Press, “A ’colossal’ ransomware attack hits hundreds of u.s. companies, a security firm says.” https://www.npr.org/2021/07/03/1012849198/ransomware-cyber-attack-revil-attack-huntress-labs.

[94] D. E. Sanger, N. Perlroth, and E. Schmitt, “Scope of russian hack becomes clear: Multiple u.s. agencies were hit.” https://www.nytimes.com/2020/12/14/us/politics/russia-hack-nsa-homeland-security-pentagon.html, 2020.

[95] S. Eggers, “A novel approach for analyzing the nuclear supply chain cyber-attack surface,” *Nuclear engineering and technology*, vol. 53, no. 3, pp. 879–887, 2021.

[96] R. Monroe, “Security.” https://xkcd.com/538/.

[97] B. Schneier, “Attack trees,” *Dr. dobb’s journal*, vol. 24, no. 12, pp. 21–29, 1999.

[98] M. Antonakakis *et al.*, “Understanding the mirai botnet,” in *26th USENIX security symposium (USENIX security 17)*, Aug. 2017, pp. 1093–1110. Available: `https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/antonakakis`

[99] K. Angrishi, “Turning internet of things (iot) into internet of vulnerabilities (iov): Iot botnets,” *Arxiv preprint arxiv:1702.03681*, 2017.

[100] B. Delamore and R. K. Ko, “A global, empirical analysis of the shellshock vulnerability in web applications,” in *2015 ieee trustcom/bigdatase/ispa*, 2015, vol. 1, pp. 1129–1135.

[101] M. Carvalho, J. DeMott, R. Ford, and D. A. Wheeler, "Heartbleed 101," *Ieee security & privacy*, vol. 12, no. 4, pp. 63–67, 2014.

[102] J. A. Kupsch and B. P. Miller, "Why do software assurance tools have problems finding bugs like heartbleed," *Continuous software assurance marketplace*, vol. 22, 2014.

[103] D. Grune, K. Van Reeuwijk, H. E. Bal, C. J. Jacobs, and K. Langendoen, *Modern compiler design*. Springer Science & Business Media, 2012.

[104] A. W. Keep and R. K. Dybvig, "A nanopass framework for commercial compiler development," in *Proceedings of the 18th acm sigplan international conference on functional programming*, 2013, pp. 343–350.

[105] C. Lattner *et al.*, "Mlir: A compiler infrastructure for the end of moore's law," *Arxiv preprint arxiv:2002.11054*, 2020.

[106] A. Munshi, "The opencl specification," in *2009 ieee hot chips 21 symposium (hcs)*, 2009, pp. 1–314.

[107] G. Balakrishnan and T. Reps, "Wysinwyx: What you see is not what you execute," *Acm transactions on programming languages and systems (toplas)*, vol. 32, no. 6, pp. 1–84, 2010.

[108] M. Chen *et al.*, "Evaluating large language models trained on code," *Arxiv preprint arxiv:2107.03374*, 2021.

[109] K. Wiggers, "Openai disbands its robotics research team." https://venturebeat.com/2021/07/16/openai-disbands-its-robotics-research-team/, 2021.

[110] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *24th $*

$USENIX$

*security symposium ($*

$USENIX$

*security 15)*, 2015, pp. 611–626.

[111] W. M. Khoo, A. Mycroft, and R. Anderson, "Rendezvous: A search engine for binary code," in *2013 10th working conference on mining software repositories (msr)*, 2013, pp. 329–338.

[112] M. R. Farhadi, B. C. Fung, P. Charland, and M. Debbabi, "Binclone: Detecting code clones in malware," in *2014 eighth international conference on software security and reliability (sere)*, 2014, pp. 78–87.

[113] S. H. Ding, B. C. Fung, and P. Charland, "Kam1n0: Mapreduce-based assembly clone search for reverse engineering," in *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, 2016, pp. 461–470.

[114] Y. David and E. Yahav, "Tracelet-based code search in executables," *Acm sigplan notices*, vol. 49, no. 6, pp. 349–360, 2014.

[115] C. Hasegawa and H. Iyatomi, "One-dimensional convolutional neural networks for android malware detection," in *2018 ieee 14th international colloquium on signal processing & its applications (cspa)*, 2018, pp. 99–102.

[116] L. Massarelli, G. A. Di Luna, F. Petroni, L. Querzoni, and R. Baldoni, "Investigating graph embedding neural networks with unsupervised features extraction for binary analysis," 2019.

[117] K. K. Thekumparampil, C. Wang, S. Oh, and L.-J. Li, "Attention-based graph neural network for semi-supervised learning," *Arxiv preprint arxiv:1803.03735*, 2018.

[118] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *International conference on machine learning*, 2016, pp. 2091–2100.

[119] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler, "Neural code comprehension: A learnable representation of code semantics," *Arxiv preprint arxiv:1806.07336*, 2018.

[120] F. Desclaux, "Miasm: Framework de reverse engineering," *Actes du sstic. sstic*, 2012.

[121] X. Wang, J. Sun, Z. Chen, P. Zhang, J. Wang, and Y. Lin, "Towards optimal concolic testing," in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 291–302.

[122] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan, "Repeatable reverse engineering with panda," in *Proceedings of the 5th program protection and reverse engineering workshop*, 2015, pp. 1–11.

[123] J. Lee, W.-S. Han, R. Kasperovics, and J.-H. Lee, "An in-depth comparison of subgraph isomorphism algorithms in graph databases," *Proceedings of the vldb endowment*, vol. 6, no. 2, pp. 133–144, 2012.

[124] J. R. Ullmann, "An algorithm for subgraph isomorphism," *Journal of the acm (jacm)*, vol. 23, no. 1, pp. 31–42, 1976.

[125] E. R. Jacobson, N. Rosenblum, and B. P. Miller, "Labeling library functions in stripped binaries," in *Proceedings of the 10th acm sigplan-sigsoft workshop on program analysis for software tools*, 2011, pp. 1–8.

[126] W. Crichton, M. Agrawala, and P. Hanrahan, "The role of working memory in program tracing," *Arxiv preprint arxiv:2101.06305*, 2021.

[127] V. M. Bertacco, *Achieving scalable hardware verification with symbolic simulation*. Stanford University, 2003.

[128] P.-H. Ho *et al.*, "Smart simulation using collaborative formal and simulation engines," in *Ieee/acm international conference on computer aided design. iccad-2000. ieee/acm digest of technical papers (cat. no. 00ch37140)*, 2000, pp. 120–126.

[129] R. Nane *et al.*, "A survey and evaluation of fpga high-level synthesis tools," *Ieee transactions on computer-aided design of integrated circuits and systems*, vol. 35, no. 10, pp. 1591–1604, 2015.

[130] V. Sklyarov, I. Skliarova, A. Barkalov, and L. Titarenko, *Synthesis and optimization of fpga-based systems*, vol. 294. Springer Science & Business Media, 2014.

[131] A. Canis *et al.*, "Legup: high-level synthesis for fpga-based processor/accelerator systems," in *Proceedings of the 19th acm/sigda international symposium on field programmable gate arrays*, 2011, pp. 33–36.

[132] "Instant soc." https://www.fpga-cores.com/instant-soc/, 2021.

[133] C. Janzen, "Autotuning the intel hls compiler using the opentuner framework." University of Saskatchewan, 2019.

[134] G. Martin and G. Smith, "High-level synthesis: Past, present, and future," *Ieee design & test of computers*, vol. 26, no. 4, pp. 18–25, 2009.

[135] C. Lavin and A. Kaviani, "Rapidwright: Enabling custom crafted implementations for fpgas," in *2018 ieee 26th annual international symposium on field-programmable custom computing machines (fccm)*, 2018, pp. 133–140.

[136] A. Izraelevitz, *Unlocking design reuse with hardware compiler frameworks*. University of California, Berkeley, 2019.

[137] R. K. Brayton, G. D. Hachtel, and A. L. Sangiovanni-Vincentelli, "Multilevel logic synthesis," *Proceedings of the ieee*, vol. 78, no. 2, pp. 264–300, 1990.

[138] W. L. Neto, M. Austin, S. Temple, L. Amaru, X. Tang, and P.-E. Gaillardon, "Lsoracle: A logic synthesis framework driven by artificial intelligence," in *2019 ieee/acm international conference on computer-aided design (iccad)*, 2019, pp. 1–6.

[139] S. Yang and M. J. Ciesielski, "Optimum and suboptimum algorithms for input encoding and its relationship to logic minimization," *Ieee transactions on computer-aided design of integrated circuits and systems*, vol. 10, no. 1, pp. 4–12, 1991.

[140] O. Coudert, "Solving graph optimization problems with zbdds," in *Proceedings european design and test conference. ed & tc 97*, 1997, pp. 224–228.

[141] M. R. Dagenais, V. K. Agarwal, and N. C. Rumin, "Mcboole: A new procedure for exact logic minimization," *Ieee transactions on computer-aided design of integrated circuits and systems*, vol. 5, no. 1, pp. 229–238, 1986.

[142] N.-C. Chou, L.-T. Liu, C.-K. Cheng, W.-J. Dai, and R. Lindelof, "Local ratio cut and set covering partitioning for huge logic emulation systems," *Ieee transactions on computer-aided design of integrated circuits and systems*, vol. 14, no. 9, pp. 1085–1092, 1995.

[143] P. Sawkar and D. Thomas, "Multi-way partitioning for minimum delay for look-up table based fpgas," in *Proceedings of the 32nd annual acm/ieee design automation conference*, 1995, pp. 201–210.

[144] R. K. Brayton, G. D. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic minimization algorithms for vlsi synthesis*, vol. 2. Springer Science & Business Media, 1984.

[145] A. Saldanha, T. Villa, R. K. Brayton, and A. L. Sangiovanni-Vincentelli, "Satisfaction of input and output encoding constraints," *Ieee transactions on computer-aided design of integrated circuits and systems*, vol. 13, no. 5, pp. 589–602, 1994.

[146] J. H. Tracey, "Internal state assignments for asynchronous sequential machines," *Ieee transactions on electronic computers*, no. 4, pp. 551–560, 1966.

[147] R. L. Rudell and A. Sangiovanni-Vincentelli, "Multiple-valued minimization for pla optimization," *Ieee transactions on computer-aided design of integrated circuits and systems*, vol. 6, no. 5, pp. 727–750, 1987.

[148] J. Cong and C. L. Liu, "On the k-layer planar subset and topological via minimization problems," *Ieee transactions on computer-aided design of integrated circuits and systems*, vol. 10, no. 8, pp. 972–981, 1991.

[149] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, "Dependence graphs and compiler optimizations," in *Proceedings of the 8th acm sigplan-sigact symposium on principles of programming languages*, 1981, pp. 207–218.

[150] A. Orailoglu and D. D. Gajski, "Flow graph representation," in *Proceedings of the 23rd acm/ieee design automation conference*, 1986, pp. 503–509.

[151] J. G. Siek, *Essentials of compilation: An incremental approach in racket*. MIT Press, 2023.

[152] C. J. Cheney, "A nonrecursive list compacting algorithm," *Communications of the acm*, vol. 13, no. 11, pp. 677–678, 1970.

[153] Y. Sui and J. Xue, "Svf: interprocedural static value-flow analysis in llvm," in *Proceedings of the 25th international conference on compiler construction*, 2016, pp. 265–266.

[154] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "Efficiently computing static single assignment form and the control dependence graph," *Acm transactions on programming languages and systems (toplas)*, vol. 13, no. 4, pp. 451–490, 1991.

[155] C. D. Offner, "Notes on graph algorithms used in optimizing compilers." 1995.

[156] R. K. Brayton, G. D. Hachtel, C. T. McMullen, and A. L. Sangiovanni-Vincentelli, "The espresso-ii minimization loop and algorithms," in *Logic minimization algorithms for vlsi synthesis*, Springer, 1984, pp. 54–138.

[157] H. Kanakia, M. Nazemi, A. Fayyazi, and M. Pedram, "Espresso-gpu: Blazingly fast two-level logic minimization," in *2021 design, automation & test in europe conference & exhibition (date)*, 2021, pp. 1038–1043.

[158] M. Ju *et al.*, "Multi-task self-supervised graph neural networks enable stronger task generalization," *Arxiv preprint arxiv:2210.02016*, 2022.

[159] F. M. Bianchi, D. Grattarola, and C. Alippi, "Spectral clustering with graph neural networks for graph pooling," in *International conference on machine learning*, 2020, pp. 874–883.

# ALEXANDER SHROYER

email = ashroyer@gmail.com

website = https://alexshroyer.com

GitHub = https://github.com/hoosieree

LinkedIn = https://www.linkedin.com/in/alexander-shroyer-82634580

## RESEARCH STATEMENT

I enjoy applying deep learning methods in novel domains. My Ph.D. research focused on classifying obfuscated compiled binary code using deep learning. This work improves forensic security analysis and software engineering quality metrics. The insights obtained from using deep learning in this niche are applicable to adjacent domains such as program synthesis, hardware synthesis, compiler optimizations, and computer aided design.

## PROFESSIONAL EXPERIENCE

*Indiana University Intelligent Systems Engineering*

**Systems Engineer** (2016-07-01 – present)

- Developed machine learning tools for obfuscated binary analysis
- Managed VMs and containers for grading software-as-a-service (SaaS), serving hundreds of concurrent students across multiple courses in 2 departments
- Built and maintained cluster, server, power, and switch infrastructure in a data center
- Built 64-node Xilinx FPGA cluster with 40G network interconnects for high performance graph analytics

- Merged 8-GPU cluster with 16-node CPU/FPGA cluster for hardware accelerated networking applications
- Created curriculum for and served as primary instructor for 2 required courses and 2 electives
- Co-taught 3 required courses
- Compiled and organized data collection for all courses for ABET accreditation

*Indiana University Psychological and Brain Sciences*

**Electronics Engineer** (2014-03-03 – 2016-06-30)

- Developed novel MRI-safe touch screen technology resulting in US Patent 10820839B2
- Created novel sensors for research: including skin response, eye tracking systems, brain-computer interfaces, pressure sensing surfaces
- Designed data acquisition hardware and software with hard-real time constraints to interface with multiple operating systems
- Helped renovate storage space into a combined electronics, carpentry, and machine shop containing CNC mills, lathes, saws, and PCB fabrication services
- Designed and built custom hardware for experimental devices

## EDUCATION

- **Intelligent Systems Engineering Ph.D**, Indiana University (2018-08-20 – 2023-12-04). Computer Engineering major, Computer Science minor. Dissertation: Deep Learning for Obfuscated Code Analysis.
- **Bachelors of Electrical Engineering**, Indiana University Purdue University Indianapolis (2010-05-11 – 2013-12-16). Electrical and Computer Engineering major.

## PUBLICATIONS

- **Shroyer, Alexander**, Paventhan Vivekanandan, and D. Martin Swany. "Function Classification for Obfuscated Binary Code". Submitted to IEEE Transactions on Information Forensics and Security, December 2023.

- **Shroyer, Alexander**, and D. Martin Swany. "Detecting Standard Library Functions in Obfuscated Code." Accepted for publication at IntelliSys2023.

- **Shroyer, Alexander**, and D. Martin Swany. "Data Augmentation for Code Analysis." 2022 International Conference on Intelligent Data Science Technologies and Applications (IDSTA). IEEE, 2022.

- Brasilino, Lucas RB, Naveen Marri, **Alexander Shroyer**, Catherine Pilachowski, Ezra Kissel, and Martin Swany. "In-network processing for edge computing with InLocus." International Journal of Cloud Computing 9, no. 1 (2020): 55-74.

- Brasilino, Lucas RB, **Alexander Shroyer**, Naveen Marri, Saurabh Agrawal, Catherine Pilachowski, Ezra Kissel, and Martin Swany. "Data Distillation at the Network's Edge: Exposing Programmable Logic with InLocus." In 2018 IEEE International Conference on Edge Computing (EDGE), pp. 25-32. IEEE, 2018.

- Arap, Omer, Lucas RB Brasilino, Ezra Kissel, **Alexander Shroyer**, and Martin Swany. "Offloading collective operations to programmable logic." IEEE Micro 37, no. 5 (2017): 52-60.

## AWARDS

Best Presentation Award, Intelligent Systems Conference (IntelliSys), September 9 2023.

## TALKS

"Thinking in Array Languages". Interviewed by Richard Feldman for the Software Unscripted podcast[1]. July 8 2023.

## PATENTS

**Electronic tablet for use in MRI.** WO US CA US10820839B2, Indiana University of Research and Technology Corporation (2020). This patent describes a system for recording visually guided motor activity using a touch-screen device within the bore of an fMRI machine while it conducts a brain scan. The device measures handwriting's impact[2] on early brain development, described in detail here[3].

## TEACHING EXPERIENCE AT INDIANA UNIVERSITY

- Engineering Computing Architecture E110 (2018 – present) primary instructor
- Engineering Networks E318/E518 (2016 – present) co-instructor
- Engineering Operating Systems E319/E519 (2016 – 2021) curriculum development
- Engineering Ethics and Professionalism E299 (2019 – present) primary instructor
- Software Systems Engineering E111 (2019 – 2021) co-instructor
- Innovation and Design E101 (2016 – 2018) co-instructor
- Python for Engineers E399/599 (2020 – present) primary instructor (elective)

[1] https://podcasts.apple.com/us/podcast/thinking-in-array-languages-with-alex-shroyer/id1602572955?i=1000619306747

[2] https://doi.org/10.1162/jocn_a_01340

[3] https://www-sciencedirect-com.proxyiub.uits.iu.edu/science/article/pii/S0165027018301560

## OPEN SOURCE SOFTWARE

- ObfuscatedBinaryClassifiers on GitHub[4].
- HDL Online[5], Source Code[6].
- Personal website[7], Source Code[8].
- Advent of Code 2022 in K[9].
- Enabling GPU support for the J programming language[10].
- PyTorch contributions[11].

## TECHNICAL SKILLS

*Proficiencies*

- NumPy, PyTorch, and Tensorflow
- Seaborn when possible, Matplotlib when necessary
- reproducible builds, Docker, docker-compose
- code obfuscation and countermeasures
- lightweight, no-framework web applications
- org-mode and static site generation
- compilers

[4] https://github.com/hoosierEE/ObfuscatedBinaryClassifiers
[5] https://alexshroyer.com/hdl/
[6] source code: https://github.com/hoosierEE/n2t-web
[7] https://alexshroyer.com/
[8] Website source code: https://gitlab.com/hoosieree/hoosieree.gitlab.io
[9] Advent of K: https://github.com/hoosierEE/aoc
[10] PDF+code: https://alexshroyer.com/papers/matmul_j_gpu.pdf
[11] https://github.com/pytorch/examples/pull/1198

*Programming Languages*

**Prefer to work with:** K, Python

**Prefer to avoid:** Verilog

**Curious to learn more about:** Elixir, Zig

My ideal language uses a hypothetical version of K (with lexical closures) as a fast execution kernel DSL, which is then wrapped in an actor-model concurrent runtime like Elixir for dispatching work to accelerators. Back in reality I use Python most often and C occasionally. While I would not go so far as to claim total mastery of either C or Python, I know more than enough to be not only dangerous, but productive.

*Deep Learning Hands-On Experience*

- Computer Vision
- Natural Language Processing
- Novel domains such as obfuscated binary program data

*Hardware*

- Hardware construction workflows: Chisel, MyHDL
- Datacenter operations
- Building and maintaining multi-user, high performance compute and accelerator cluster hardware