# Graph Embeddings for the extraction of Compiler Provenance features

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Aleksandar N. Straumann

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Peter. A. H. Peterson

June 2019

© Aleksandar N. Straumann 2019

## Acknowledgements

Firstly, I would like to express my sincere gratitude to my advisor Professor Peter Peterson for the continuous support of my research, for his patience, motivation, and immense knowledge. His guidance helped me in all the time of research and writing of this thesis. I could not have imagined having a better advisor and mentor for my masters. Besides my advisor I would also like to thank my mentor at Sandia National Labs, Michael Bierma, for offering me guidance throughout this process. Sandia National Laboratories is a multi-mission laboratory managed and operated by National Technology  Engineering Solutions of Sandia, LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.

Abstract

This thesis explores the use of graph embedding methods for compiler provenance identification. Graph embedding algorithms are widely used to analyze, compare, or distinguish networks, or similar structures, that are too large to represent visually. Using graph embeddings to address the problem of compiler provenance identification is a novel approach. Our approach applies embedding algorithms to the control flow graphs of binaries. In this document, we explore two graph embedding methods: tiered approaches and alternative embedding representations for analysis. Our results indicate that our method has the potential for use in compiler provenance identification. Experiments show that our approach is able to distinguish between individual compilers, compiler versions, and compiler version flags with above-average accuracy. Future work may explore extracting the significant graph embeddings from our generated model, recreate the generalized graph from the embeddings, and identify significant structures for manual analysis.

# Contents

**Contents** iii

**List of Tables** v

**List of Figures** vii

**1 Introduction** 1

**2 Related Works** 3

2.1 Background . . . 3

2.1.1 Compiler Provenance . . . 3

2.1.2 Static Analysis . . . 5

2.1.3 Graph Embeddings . . . 11

**3 Implementation** 15

3.0.1 Disassembly . . . 15

3.0.2 Control Flow Graph . . . 17

3.0.3 Graph Embedding Generation . . . 18

3.0.4 Padding . . . 19

3.0.5 Machine Learning . . . 21

**4 Results** 22

4.0.1 Data Generation . . . . . . . . . . . . . . . . . . . . . . . . 22

4.0.2 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 24

4.0.3 Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . 31

4.0.4 Discussion . . . . . . . . . . . . . . . . . . . . . . . . . . . . 32

**5 Conclusions** 36

5.0.1 Future Works . . . . . . . . . . . . . . . . . . . . . . . . . . 36

5.0.2 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38

**References** 39

# List of Tables

4.1 The compilers, their versions, and the optimizations applied for the binaries in our project. . . . . . . . . . . . . . . . . . . . . . . . . . . 24

4.2 Classification of the Compilers using HOPE Embeddings with a Tiered approach. There is a 50% chance of classifying correctly through random chance; better than 50% indicates that the method is successful. 24

4.3 Classification of the Version assuming known Compiler using HOPE Embeddings with a Tiered approach. (33% random chance of correct classification.) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 25

4.4 Classification of the Optimizations assuming known Compiler and Version using HOPE Embeddings with a Tiered approach. (20% random chance of correct classification.) . . . . . . . . . . . . . . . . . . . . . 25

4.5 Classification of Compiler and Version using HOPE Embeddings with a Non-Tiered approach. (15% random chance of correct classification.) 25

4.6 Classification of Compiler and Version and Optimization using HOPE Embeddings with a Non-Tiered approach. (3% random chance of correct classification.) . . . . . . . . . . . . . . . . . . . . . . . . . . 26

4.7 Classification of Compilers using HOPE alongside Average Length Padding in a Tiered approach. (50% random chance of correct classification.) . . . . . 26
4.8 Classification of Version knowing Compiler using HOPE Embeddings alongside Average Length Padding with a Tiered approach. (33% random chance of correct classification.) . . . . . 27
4.9 Classification of Compiler, Version and Optimizations using HOPE Embeddings alongside Average Length Padding with a Non-Tiered approach. (3% random chance of correct classification.) . . . . . 28
4.10 Classification of Compiler, Version and Optimizations using Node2Vec Embeddings with a Non-Tiered approach. (3% random chance of correct classification.) . . . . . 29

# List of Figures

2.1 Compiler source to machine code translation . . . . . . . . . . . . . . 4
2.2 Program . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8
2.3 Layout obfuscation . . . . . . . . . . . . . . . . . . . . . . . . . . 8
2.4 Hello World C . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 9
2.5 Hello World Disassembly . . . . . . . . . . . . . . . . . . . . . . . 9
2.6 Hello World Obfuscated . . . . . . . . . . . . . . . . . . . . . . . . 10
2.7 Data Obfuscation . . . . . . . . . . . . . . . . . . . . . . . . . . . 11
2.8 Control Flow Graph . . . . . . . . . . . . . . . . . . . . . . . . . . 13
2.9 HOPE . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14
2.10 Node2Vec . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 14

3.1 ELF Headers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.2 Program Headers . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.3 Basic Block Main . . . . . . . . . . . . . . . . . . . . . . . . . . . 17
3.4 Column wise . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 20

4.1 Dockerfile . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
4.2 make.conf . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23

4.3 This is the visualization of the data shown in Table 4.4. Each column is the optimization applied, each symbol-color combination represents a compiler and version. (20% random chance of correct classification.) 26

4.4 Visualization of the data shown in Table 4.6. Each symbol corresponds to a compiler and version and the accuracy predicting the flags used for compilation. (3% random chance of correct classification.) . . . . 27

4.5 Visualization of the data shown in Table 4.9. Each symbol corresponds to a compiler and version and the accuracy predicting the flags used for compilation. (3% random chance of correct classification.) . . . . 28

4.6 Visualization of the data shown in Table 4.10. Each symbol corresponds to a compiler and version and the accuracy predicting the flags used for compilation. (3% random chance of correct classification.) . . 29

4.7 Boxplots displaying the accuracy of each approach for each flag. Red shows tiered accuracy. Blue shows the Non-tiered accuracy. Purple highlights the average padding approach accuracy. Green displays the accuracy using Node2Vec embeddings. Diamonds indicate outliers. The random chance of success for Tiered was 20%, while for all other methods it was 3%. . . . . . . . . . . . . . . . . . . . . . . . . . . . . 30

# 1 Introduction

Reverse engineering software takes a finalized application, disassembles it, and inspects its parts to learn more about the architecture, coding style, authorship, or its functionality. This often requires that the individual attempting to reverse engineer has incredibly detailed knowledge on a myriad of subjects including, but not limited to: operating systems, compiler design, programming, architecture design, and programming languages. Often, reverse engineers attempt to redevelop the original source from the machine code, but this is difficult due to the compilation process that produces an application. Compilation takes source code and converts it to machine code, removing information useful to humans, such as syntax, names, or comments. Further complicating the process is the fact that individuals compiling software can obfuscate or maliciously change their application to hide information so that reverse engineering is much more difficult. Thus, it is important to have tools that can aid an individual in the task of reverse engineering by identifying information that would be helpful for proper reverse engineering.

To effectively reverse engineer an application, an engineer needs to know what computer architecture the application was built for, the compiler used, its version, and potentially the optimizations used during compilation. Collectively these are referred to as compiler provenance features and they are extremely important due to the compounding effect that misidentifying these features may have. Extracting compiler provenance from an application is done using heuristics, pre-built models, signatures, or other patterns that identify specific features in an application that

indicate some element of the application's compiler provenance. An engineer may make use of tools that contain databases of these signatures in order to present the most likely compiler provenance features. However, signatures are not always accurate and can present false positives.

Misidentifying any one of these features can lead to the inability to successfully apply various binary analysis techniques (Rahimian et al. 2015). As mentioned above, compilation is a one-way process; to undo it, reverse engineers make the best guess of compiler provenance features to accurately disassemble and interpret the underlying machine code. Any conclusions drawn from incorrectly reverse engineered programs may be incorrect because they are based on false premises; machine code can differ drastically from any compiler, version, or optimization made. Thus, it is critical for compiler provenance extraction and identification to be accurate.

This thesis presents a novel and generalized approach to compiler provenance identification. This work takes approaches found in similar research done by R. Wang et al. 2017 and S. Wang, P. Wang, and Wu 2016 in disassembly and control flow graph (CFG) construction. Unlike prior works, our work uses a low dimensional representation of CFGs, namely graph embeddings, for compiler provenance detection. In short, we use simple disassembly methods to develop CFGs and run two separate graph embedding algorithms against the developed graphs. Graph embeddings, which reduce the high dimensionality of large graphs into lower dimensional representation, are applied to the CFGs which can then be used for classifcation. Once gathered, the graph embeddings are run through a a machine learning classifier for compiler provenance prediction.

In the rest of this paper we will discuss the related works, implementation, and results of our experiments.

# 2 Related Works

## 2.1 Background

### 2.1.1 Compiler Provenance

Compiler provenance is a term that encompasses many details of the compiler used to create a particular binary: the compiler family, version, optimization levels, and related functions (Rahimian et al. 2015). These features are pivotal for understanding a binary and its results. However, extraction of this information can be very difficult due to the asymmetrical nature of a compiler.

Compilers take source code, written in higher level languages, and convert them into machine code. The equivalent machine code is generated using specific rules. This process of compilation is a one way transmutation, as much of the information that makes source code easy for humans to debug is stripped away. As such, reading machine code is difficult due to the lack of user defined variable names, offsets, and branching that may have occurred in the source code. When reading machine code it is also important to note that sections of relevant data are intermixed with other information, making it impossible to read linearly. Additionally, branching factors such as function calls, switches, and obfuscation techniques may change offsets and other sections of information. Finally, machine code often makes use of libraries which are accessed through the linker; linking allows a program to make calls to other functions that are not stored in the binary itself, further obfuscating the process.

Compiler provenance allows a reverse engineer the ability to distinguish bits of machine code as compiler-specific or not. Compiler specific machine code is assembly code generated from a given set of rules such as loop unpacking, encryption algorithms, and other special cases, knowing which compiler generated the machine code allows a reverse engineer to quickly pinpoint and discard unnecessary machine code that won't aid in their task. Compiler family (gcc, clang, etc.) and versions (gcc-5, clang-6, etc.) allow an engineer to understand what compiler instructions will be generated (Krügel et al. 2004). Optimization level allows an engineer to understand and find common patterns generated when optimizing machine code. These all allow for an accurate recreation of a binary's source. Without exact extraction of this information, accurate reproduction of any source becomes impossible. Thus it is important for reverse engineers to be able to gather this information.

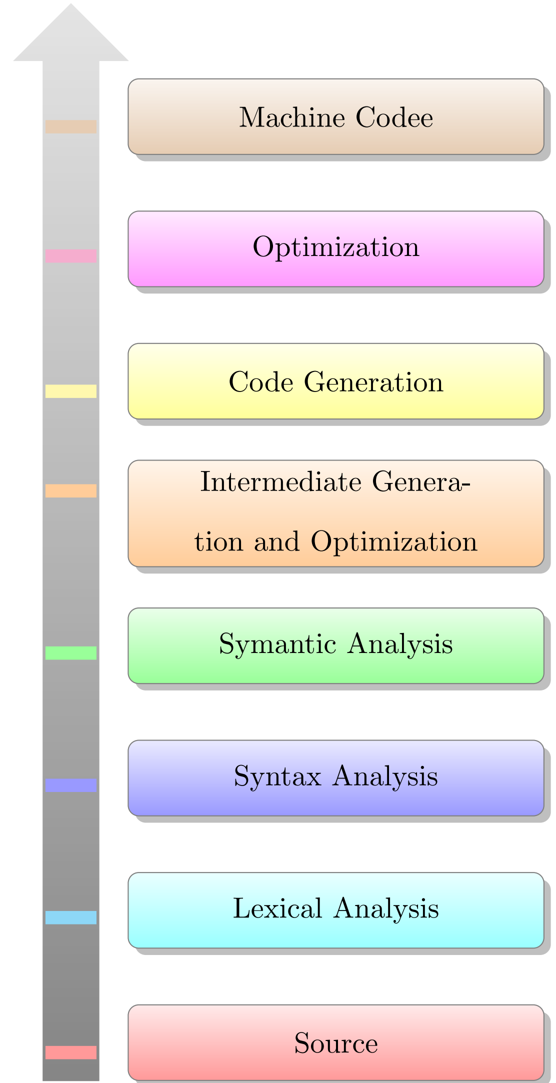


Figure 2.1: Compiler source to machine code translation

In the past decade there have been few notable studies such as early work done by Jacobson, N. Rosenblum, and Miller 2011, "Labelling Library Functions in Stripped Binaries", which focuses only on semantic descriptors and highlights modern issues in compiler provenance extraction such as: lack of types, expressive syntax, comments, formatting, and compiler transforma-

tions (Jacobson, N. Rosenblum, and Miller 2011). Semantic descriptors are later expanded upon and used in works by Rosenblum's, Miller's, and Zhun's work. These descriptors are used in identifying function entry points, another common issue when dealing with compiler provenance extraction.

"Extracting Compiler Provenance in Stripped Binaries" by Rosenblum, Miller, and Zhun (ECP), takes an N-Gram like approach to feature extraction. Commonly used in Natural Language Processing, N-grams look at surrounding neighbors and develops pairs or sequences of words. ECP uses an improved N-gram-like approach which they refer to as "idioms". Idioms are developed by taking short sequences of instructions and optional wildcards (N. E. Rosenblum, Miller, and Zhu n.d.). Idioms are extremely useful because they capture instruction patterns, which can be related back to bytes in the binaries themselves.

BinComp is the most recent approach to compiler provenance extraction, and in contrast to its predecessors, takes a more comprehensive approach to feature extraction. BinComp breaks the problem into three layers: syntax extraction, compiler function features, and semantic features (Rahimian et al. 2015). This approach relies on a large dataset containing source, binary, and all intermediary files produced during compilation. This process results in a robust data set, improving overall accuracy (Rahimian et al. 2015).

### 2.1.2 Static Analysis

Disassembly is a fundamental component of both reverse engineering and compiler provenance identification. Disassembly is incredibly difficult due to the many improvements made to compilers, and obfuscation techniques. Failure to disassemble target binaries correctly may result in a cascading series of problems. Problems

during disassembly can result in inaccurate code reproduction, error prone patching, misclassification of code while wasting large amounts of time. There have been extensive efforts in creating tools that have high accuracy when identifying these features for disassembly.

These tools are referred to as disassemblers, and there are a plethora of tools available for use, such as: IDA Pro (Hex-Rays 2019), Binary Ninja (Vector-35 2019), Radare2 (R. Wang et al. 2017), and Ghidra (NSA 2019). These tools utilize metadata included in target binaries to aid in their task. Others have developed signatures that can aid in the accuracy of the disassembly as well. However, one of the most basic features of any disassembler is to provide a dump of instructions included in the target.

There are two main approaches when disassembling binaries: Linear Sweep and Recursive Traversal (R. Wang et al. 2017). Linear sweep starts with the first byte of the target's text section and begins disassembly from that byte (Krügel et al. 2004). There are downsides to this approach; one such issue is the inability to distinguish non-data sections from data (Krügel et al. 2004). Recursive traversal avoids this problem by following addresses available in disassembly that affect control flow. The recursive approach then begins disassembling from these new addresses (Meng and Miller 2016). The recursive approach is more likely to avoid missing sections of code as it will iterate over sections more than once, however, it has its own set of problems. Such problems come in the form of effectiveness, efficiency, and the implementation of the recursive method. As a recursive method by definition goes through the code multiple times until it has seen all possible addresses there is the issue of potentially accessing areas of memory that are not strictly machine code. This can lead to further disassembly of invalid sections of memory, which can compound into a completely irrelevant disassembly of the application. Furthermore, while linear methods are

simple and start inside the start point of the application, recursive methods will be much more likely to fall victim to obfuscation techniques, simply because obfuscation techniques often edit control flow, names, or simply add code which can make recursive disassembly less effective. As mentioned, obfuscation techniques can be used to limit the efficacy of both recursive and linear methods (Krügel et al. 2004). However, recursive methods are much more likely to suffer due to their multiple passes through the machine code. Linear and recursive methods are not binary however, and it is often the case that they are used in conjunction to more effectively disassemble binaries. Finally, it is worth mentioning that obfuscation can be overcome, however, the methods to do so are often much more involved than simple linear or recursive methods (Schwarz, S. Debray, and G. Andrews 2002).

A sufficiently skilled and committed individual will eventually reverse engineer their target. However, obfuscation techniques are often used to inhibit and deter potentially malicious individuals from reverse engineering a given program (Popov, S. K. Debray, and G. R. Andrews 2007). Obfuscation techniques have been created to hinder a variety of approaches to reverse engineering(Hosseinzadeh et al. 2018): Layout obfuscation randomly adjusts the source code in an attempt to make disassembly difficult ( Figure 2.4,Figure 2.2,Figure 2.3) (ioccc 2019, Hosseinzadeh et al. 2018, Popov, S. K. Debray, and G. R. Andrews 2007). Control flow obfuscation changes the control flow of the program into a convoluted mess (Figure 2.5 Figure 2.6) (Hosseinzadeh et al. 2018, Popov, S. K. Debray, and G. R. Andrews 2007). Finally, data obfuscation hides data structures, types, and variables in a program (Figure 2.4,Figure 2.7) (Hosseinzadeh et al. 2018, Popov, S. K. Debray, and G. R. Andrews 2007). Data and layout obfuscation may seem similar, however layout obfuscation focuses on source code while data obfuscation seeks to obscure the function calls in both source and the underlying disassembly that it produces.

```
1 #include <stdio.h>
2 int some_math(int a, int b){
3     int c = 8;
4     int d = a * c;
5     printf("%d", d);
6     d = d / b;
7     return d;
8 }
```

Figure 2.2: Program
Simple C program that takes in two values and does some basic arithmetic. Layout obfuscation is much more effective with more information, hence the addition of more instructions as compared to the hello world program shown below in Figure 2.4

```
1 #include <stdio.h>
2 int o_252b41c8d82534efa339ce5a5eb167c8(int
    o_8a65c6831e60fcc93ace8d2548a45b0f,int
    o_12c4c5ddbecbe57d3259e214258eb0e3){int
    o_4ce9740e66b227d830a07ac04db2216d=(0x0000000000000010 + 0
    x0000000000000208 + 0x0000000000000808 - 0x0000000000000A18)
    ;int o_631bd8e30e3054c3d26bf21e78ee6f60=
    o_8a65c6831e60fcc93ace8d2548a45b0f *
    o_4ce9740e66b227d830a07ac04db2216d;printf("\x25""d",
    o_631bd8e30e3054c3d26bf21e78ee6f60);
    o_631bd8e30e3054c3d26bf21e78ee6f60 =
    o_631bd8e30e3054c3d26bf21e78ee6f60 /
    o_12c4c5ddbecbe57d3259e214258eb0e3;return
    o_631bd8e30e3054c3d26bf21e78ee6f60;};
```

Figure 2.3: Layout obfuscation
Example of layout obfuscation applied to code in Figure 2.2. As mentioned the variable are the only layer affected, meaning with some small amount of effort one could translate this back to its more readable format.

Obfuscation techniques present a difficult problem for reverse engineers. If done well, these techniques can be stealthy, avoiding detection by manual inspections (Popov, S. K. Debray, and G. R. Andrews 2007). Regardless, there is a trade-off between obfuscation effectiveness and run time as obfuscation generally increases run time (Behera and Bhaskari 2015). Inevitably, this trade-off prevents obfuscation techniques from becoming too much of a burden for reverse engineers. In addition to this trade-off, the modification of underlying binary code can tip off engineers to the use

```c
#include <stdio.h>
int main() {
    printf("Hello World!");
}
```

Figure 2.4: Hello World C
Simple hello world program written in C for use in illustrating the obfuscation techniques shown in Figure 2.5 and Figure 2.6.

Figure 2.5: Hello World Disassembly
Unobfuscated disassembly of hello world program shown in Figure 2.4. Using the IDA disassembler to show what the main program is translated to.

of obfuscation techniques (Taylor 2017). Automated scanners are regularly used to increase the efficiency of reverse engineering, they can also be used to detect obfuscation techniques and flag them for manual inspection. Obfuscation techniques cannot prevent reverse engineering; they only increase the time and effort required of the reverse engineers.

As is often the case, reverse engineers use a mix of techniques when taking apart a binary. Rarely are engineers left with just binary disassembly to review. Often, their tools, such as disassemblers, provide a suite of techniques and representations

```
; Segment type: Pure code
; Segment permissions: Read/Execute
_text segment byte public 'CODE' use32
assume cs:_text
;org 804825Ch
assume es:nothing, ss:nothing, ds:_data, fs:nothing, gs:nothing


public _start
_start proc near

var_200068= dword ptr -200068h
arg_0= dword ptr  4
arg_4= dword ptr  8

mov     sesp, esp
mov     esp, off_83F4110
mov     esp, [esp+var_200068]
mov     esp, [esp+var_200068]
mov     esp, [esp+var_200068]
mov     esp, [esp+var_200068]
mov     dword ptr [esp+0], 0Bh
mov     [esp+arg_0], offset sa_dispatch
mov     [esp+arg_4], 0
call    _sigaction
mov     esp, off_83F4110
mov     esp, [esp+var_200068]
mov     esp, [esp+var_200068]
mov     esp, [esp+var_200068]
mov     dword ptr [esp+0], 4
mov     [esp+arg_0], offset sa_loop
mov     [esp+arg_4], 0
call    _sigaction
```

```
public master_loop
master_loop:            ; Alternative name is 'master_loop'
mov     esp, off_83F4110 ; _start0
mov     eax, toggle_execution
mov     eax, sel_on[eax*4]
mov     dword ptr [eax], 1
mov     toggle_execution, 0
mov     eax, sesp
mov     edx, 4
mov     alu_x, eax
mov     alu_y, edx
mov     eax, 0
mov     ecx, 0
mov     alu_c, 0
mov     ax, word ptr alu_x
mov     cx, word ptr alu_y
mov     edx, alu_add16[eax*4]
mov     word ptr alu_s, dx
mov     alu_c, edx
mov     ax, word ptr alu_x+2
mov     cx, word ptr alu_y+2
mov     edx, alu_add16[eax*4]
mov     edx, [edx+ecx*4]
mov     cx, word ptr alu_c+2
mov     edx, alu_add16[edx*4]
mov     edx, [edx+ecx*4]
mov     word ptr alu_s+2, dx
mov     alu_c, edx
mov     eax, alu_s
mov     eax, eax
mov     stack_temp, eax
mov     eax, offset off_83F4110
mov     edx, on
mov     data_p, eax
mov     eax, sel_data[edx*4]
mov     edx, off_83F4110
mov     edx, [edx-200068h]
mov     [eax], edx
mov     eax, off_83F4110
mov     edx, on
mov     data_p, eax
mov     eax, sel_data[edx*4]
mov     edx, stack_temp
mov     [eax], edx
mov     eax, sesp
mov     eax, [eax]
mov     eax, eax
mov     stack_temp, eax
mov     eax, offset off_83F4110
mov     edx, on
mov     data_p, eax
mov     eax, sel_data[edx*4]
mov     edx, off_83F4110
mov     edx, [edx-200068h]
mov     [eax], edx
mov     eax, off_83F4110
mov     edx, on
mov     data_p, eax
mov     eax, sel_data[edx*4]
mov     edx, stack_temp
mov     [eax], edx
mov     eax, 8804853Eh
mov     eax, eax
mov     stack_temp, eax
mov     eax, offset off_83F4110
mov     edx, on
mov     data_p, eax
mov     eax, sel_data[edx*4]
mov     edx, off_83F4110
mov     edx, [edx-200068h]
mov     [eax], edx
mov     eax, off_83F4110
```

Figure 2.6: Hello World Obfuscated

Obfuscation of the hello world program shown in figure Figure 2.4. This illustrates an example of control flow obfuscation, translating what would look like Figure 2.5 and making it much more convoluted.

```
1  #include "stdio.h"
2  _(__,_0,O_,___){(O_<_0)?printf("%c",(O_==2)
3  ||(O_==3)||(O_==9)?___++,O_++,108:*((int*)__
4  +O_++--___)),_(__,_0, O_, ___):0;}main(){int array
5  []={72,'e',111,040,0127,0x6F,'r',0x64,041 };_(array,12,0,0);}
```

Figure 2.7: Data obfuscation
Example of data obfuscation applied to the code found in Figure 2.2. Similar to the layout obfuscation technique shown in Figure 2.3, this has the added effect of altering how the data is referenced in the assembly.

that can be applied to a given binary (Hex-Rays 2019). One such representation provided are Control Flow Graphs (CFG) which visualize the underlying control flow of the disassembly (Xu, Sun, and Su 2010). This visualization is ideal for manual and automated analysis due to the strict rules associated with CFG construction. Such analysis can be used in areas outside of reverse engineering such as generating test cases (Gotlieb, Botella, and Rueher 1998), finding software bugs (Bonfante, Kaczmarek, and Marion 2007), and program verification (Gotlieb, Botella, and Rueher 1998). In addition to these methods, CFGs can be used to identify algorithms and data structures, allowing reverse engineers to easily identify possible areas of interest (Mikhailov et al. 2016).

### 2.1.3 Graph Embeddings

Graph embeddings are a product of graph analysis techniques that aim to reduce the dimensionality of graphs down to a set of one or more vectors (Ou et al. 2016). These embeddings have many practical uses for representing graphs that may be useful in graph analytics such as node classification, link prediction, clustering, and visualization (Goyal and Ferrara 2017). Node classification can help us categorize nodes based off others in a graph. Link prediction focuses on accurately predicting future connections in a more dynamic graph. Clustering, similar to node classifi-

cation, looks to group nodes based off various information. Finally, visualization is an attempt to simplify graphs into a format which we can extract more information from.

A recent survey developed a taxonomy of methods for graph embeddings. According to them (Goyal and Ferrara 2017) factorization based methods represent graphs as a matrix of node connections to obtain an embedding, but these methods do not learn structural equivalence unless included explicitly (Goyal and Ferrara 2017). Random walk based methods take partial representation of graphs via random walks, in order to represent the overall larger graph, which can model a variety of functions depending on their parameters (Goyal and Ferrara 2017). Finally, deep learning methods look at applying autoencoders to capture non-linear structures and can model a large variety of functions (Goyal and Ferrara 2017). Each of these methods focuses on extracting a representation of the N-dimensional graphs such that we can represent them in other ways. Each of the methods (Factorization, Random Walk, and Deep Learning) will produce different results and their use is dependant on the type of data being processed.

Graph embedding take highly dimensional graph data and create a lower dimensional representation called an embedding. For example, Figure 2.8 provides us with a snippet of a CFG. There are no weights or other information, outside of neighboring nodes, that can be used for tasks outlined above. HOPE is an algorithm developed by Ou et al. 2016, which focuses on preserving higher-order proximity in graph embeddings and is an example of a factorization based method. Order refers to the number of vertices used to represent the substructure, thus HOPE seeks to use more vertices to improve the embeddings it produces. Figure 2.9 illustrates the use of the HOPE algorithm to embed a directed graph such as in Figure 2.8 into a 2D space. Random walk based methods such as Node2Vec take a different approach.

Figure 2.8: Control Flow Graph
Control flow graphs (CFG) are one way to visualize a programs function. This image illustrates an example of the CFG that we will use. Each node in the graph corresponds to a basic block which is contiguous sets of instructions which may hold a branching instruction at the end. The links correspond to the branching instructions that may be at the end of the block which points to the next block for exectuion.

Unlike HOPE, Node2Vec uses random walks from all nodes such that it can develop its embedding(Grover and Leskovec 2016). An example of Node2Vec run against the same graph as HOPE can be seen in Figure 2.10. Deep learning methods take a similar approach to machine learning methods of the same name with many hidden layers. For our purposes we decided to focus on the more direct methods of graph embeddings for our project.

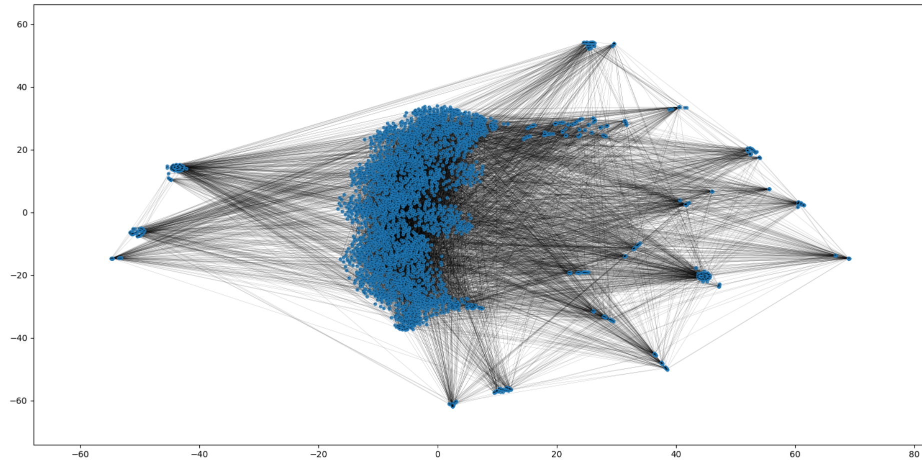


Figure 2.9: HOPE
This figure is an example of the resulting lower dimensional embedding using HOPE applied to a graph similar to Figure 2.8. This resulting embedding is the same one we use in the following approach discussed further in this paper.

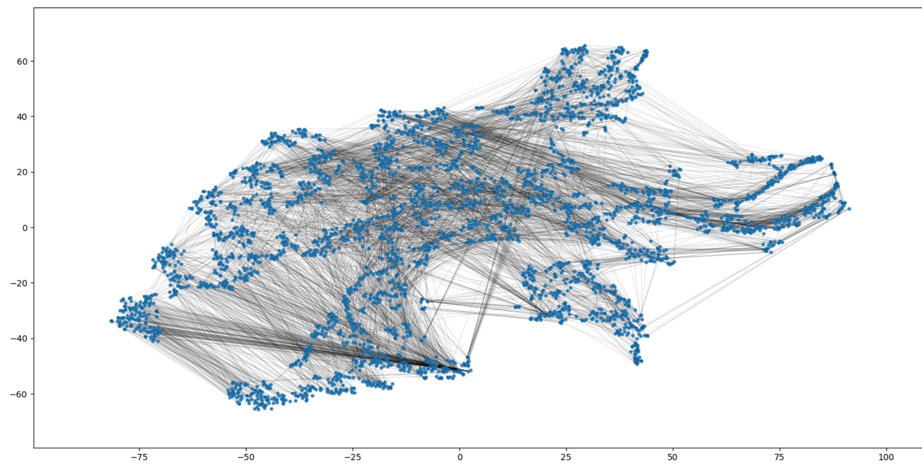


Figure 2.10: Node2Vec
This figure is an example of the resulting lower dimensional embedding using Node2Vec applied to a graph similar to Figure 2.8. This resulting embedding is the same one we use in the following approach discussed further in this paper.

# 3 Implementation

In this section we will discuss the implementation of our experiments. In this paper we take two different embedding algorithms and apply them to our generated CFGs: Node2Vec and HOPE. These methods allow us to contrast the effectiveness of two seperate graph embeddings approaches: Random Walk and Factorization. Furthermore, we use three methods for the experiments below: Tiered, Non-Tiered, and Average Length Padding. Tiered assumes prior knowledge while Non-Tiered does not. Average Length Padding, which is used in both a Tiered and Non-Tiered approach, truncates embeddings for optimization efforts. The following section discusses the implementation of these experiments.

### 3.0.1 Disassembly

Prior to the analysis, we disassemble a binary in order to generate the data for the various appraoches: Tiered, Non-Tiered, Node2Vec, and Average Length Padding. We make use of a lightweight Python module for disassembly, "capstone," [1] allowing us to have more fine-grain control over how we disassemble a binary. In our approach, we focus on disassembling only from the entry point, found in the ELF headers Figure 3.1, and subsequent addresses that we may find. In order to guarantee that we disassemble correctly, we must also determine the offset used for a given binary. The offset allows a binary to keep track of its alignment in memory, this is important

[1] https://www.capstone-engine.org/lang$_{p}ython.html$

```
ELF Header:
  Magic:   7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF64
  Data:                              2s complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Advanced Micro Devices X86-64
  Version:                           0x1
  Entry point address:               0x4049a0
  Start of program headers:          64 (bytes into file)
  Start of section headers:          124728 (bytes into file)
  Flags:                             0x0
  Size of this header:               64 (bytes)
  Size of program headers:           56 (bytes)
  Number of program headers:         9
  Size of section headers:           64 (bytes)
  Number of section headers:         29
  Section header string table index: 28
```

Figure 3.1: ELF Headers
Elf Headers are found in the beginning bytes of every ELF program. This information is necessary for the execution of the program and also dictates what size addresses will be used. For our purposes we extract the entry point address to begin our disassembly.

```
  Program Headers:
Type           Offset             VirtAddr           PhysAddr
               FileSiz            MemSiz              Flags  Align
PHDR           0x0000000000000040 0x0000000000400040 0x0000000000400040
               0x00000000000001f8 0x00000000000001f8  R E    8
INTERP         0x0000000000000238 0x0000000000400238 0x0000000000400238
               0x000000000000001c 0x000000000000001c  R      1
    [Requesting program interpreter: /lib64/ld-linux-x86-64.so.2]
LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
               0x000000000001da64 0x000000000001da64  R E    200000
LOAD           0x000000000001de00 0x000000000061de00 0x000000000061de00
               0x0000000000000800 0x0000000000001568  RW     200000
DYNAMIC        0x000000000001de18 0x000000000061de18 0x000000000061de18
               0x00000000000001e0 0x00000000000001e0  RW     8
NOTE           0x0000000000000254 0x0000000000400254 0x0000000000400254
               0x0000000000000044 0x0000000000000044  R      4
GNU_EH_FRAME   0x000000000001a5f4 0x000000000041a5f4 0x000000000041a5f4
               0x0000000000000804 0x0000000000000804  R      4
GNU_STACK      0x0000000000000000 0x0000000000000000 0x0000000000000000
               0x0000000000000000 0x0000000000000000  RW     10
GNU_RELRO      0x000000000001de00 0x000000000061de00 0x000000000061de00
               0x0000000000000200 0x0000000000000200  R      1
```

Figure 3.2: Program Headers
Program Headers follow the ELF headers and dictate how the binary will be mapped in memory. Of note are the two included LOAD sections. Once a program is mapped to memory it is split into multiple mappings so that certain areas may be READ ONLY.

as a binary will not be loaded into the same space in memory every time. This information can be determined from the information found in the program headers Figure 4.1. Once we extract these individual pieces we can begin to disassemble from the entry point, allowing us to build the CFG.

```
1        ;-- rip:
2        / (fcn) entry0 42
3        |   entry0 ();
4        |           0x004049a0      31ed            xor ebp, ebp
5        |           0x004049a2      4989d1          mov r9, rdx
6        |           0x004049a5      5e              pop rsi
7        |           0x004049a6      4889e2          mov rdx, rsp
8        |           0x004049a9      4883e4f0        and rsp, 0xfffffffffffffff0
9        |           0x004049ad      50              push rax
10       |           0x004049ae      54              push rsp
11       |           0x004049af      49c7c0203c41.   mov r8, 0x413c20          ; '<A'
12       |           0x004049b6      48c7c1b03b41.   mov rcx, 0x413bb0
13       |           0x004049bd      48c7c7002a40.   mov rdi, main             ; section..text ; 0
           x402a00 ; "AWAVAUATUS\x89\xfbH\x89\xf5H\x81\xec\x88\x03"
14       |           0x004049c4      e877dcffff      call sym.imp.__libc_start_main
15       \           0x004049c9      f4              hlt
```

Figure 3.3: Basic Block
The above shows disassembly shows one basic block. A basic blocked contains one entry point, the first instruction of the block, and can only exit at the end. In this basic block there are no branches after the last instruction, this results in a leaf node in a control flow graph which visualizes the basic block connections.

### 3.0.2 Control Flow Graph

CFGs are an important tool for the analysis of the control flow of a binary and its utilization. Our approach takes the disassembly from the first step and builds simple basic blocks from them. Basic blocks contain all instructions between branching instructions; each node in our graph represents one basic block. A small result of our CFG generation can be seen in Figure 2.8. This graph has disconnected parts due to the nature of our linear sweep approach. Linear sweep continues disassembling until it cannot properly disassemble; this results in segments of code that may never be reached by the main method of our binary during static analysis. Often when beginning disassembly; the entry point of our program may not be at the lowest point in memory, as most sections of code are not all sequential, which can cause us to miss sections of code earlier in the program.

### 3.0.3 Graph Embedding Generation

Once the CFGs (Figure 2.8) have been generated, we can begin applying specific graph embedding algorithms to them. Current embeddings algorithms accept an edge adjacency list which simply defines a node and its edges. Once we have the complete adjacency list we can begin feeding our input to the HOPE embedding and Node2Vec algorithms, saving each embedding in a its own file. For both embedding algorithms, we determined that the resulting embedding's dimensions should be of three dimensions. The embedding size was chosen to preserve as much information as possible without the resulting feature set being too large. Our chosen size of three dimensional embeddings allows us to effectively create large embeddings without requiring too much time to generate. A large part of embedding information is the trade off of information, as the operation of embedding can lose information.

The HOPE algorithm attempts to preserve higher-order proximity within graphs (Ou et al. 2016). Higher-order refers to the number of neighbors or vertices that are taken into account when developing the embedding. The HOPE algorithm takes this embedding one step further by producing an approximation method for developing a higher-order embedding, allowing us to use much larger graphs with the HOPE algorithm. The benefit of this is that when handling large graphs we can approximate an embedding, allowing us to develop models for learning and by extracting embedded features and derive the general structure from it (Goyal and Ferrara 2017).

To add some comparison for our embeddings we chose the Node2Vec algorithm to develop further embeddings for our approach. Node2Vec takes a biased random walk of our graph to develop low dimensional embeddings. Unlike HOPE, Node2Vec does not look at preserving higher-order proximity, but instead the general neighborhood in the graph. What this means is that it is unlikely to visit the same nodes in the

graph in the same order on subsequent runs, thus the biased nature of Node2Vec. The resulting embeddings from Node2Vec will have a high guarantee that all nodes in the graph have been visited but unlikely in the same order, resulting in minor changes to the embeddings.

### 3.0.4 Padding

Graph embedding algorithms produce embeddings with lengths dependant on the original graph structure. Our approach generates embeddings that are three dimensionsal. However, to effectively classify our data using machine learning many methods require that the information be coerced into a one-dimensional equivalent length format. Thus, in addition to truncating or adding padding, we must also flatten the graph embedding into a single array of values.

Padding can be an issue as artificially adding padding to our data could skew resulting data later in our experiments. To resolve this issue we apply zero padding to all embeddings to make them equivalent length. Zero in both HOPE and Node2Vec represent a lack of connection between nodes in the embedding. Thus any zeroes we add as padding will not affect the meaning of the embeddings.

Flattening of the resulting embedding is also important as most machine learning approaches require that all data is of a single dimension. For our purposes, we use column-wise flattening to retain the nodes vector representation in an embedding. Column-wise flattening is a simple method to preserve the multidimensional data in order. To flatten, we simply read, starting from the top of the column down, into a single array. This also helps to prevent data being skewed from any padding applied to the embeddings. In addition with skew prevention, there is the added benefit of being able to distinguish entire nodes from any given point selected during our

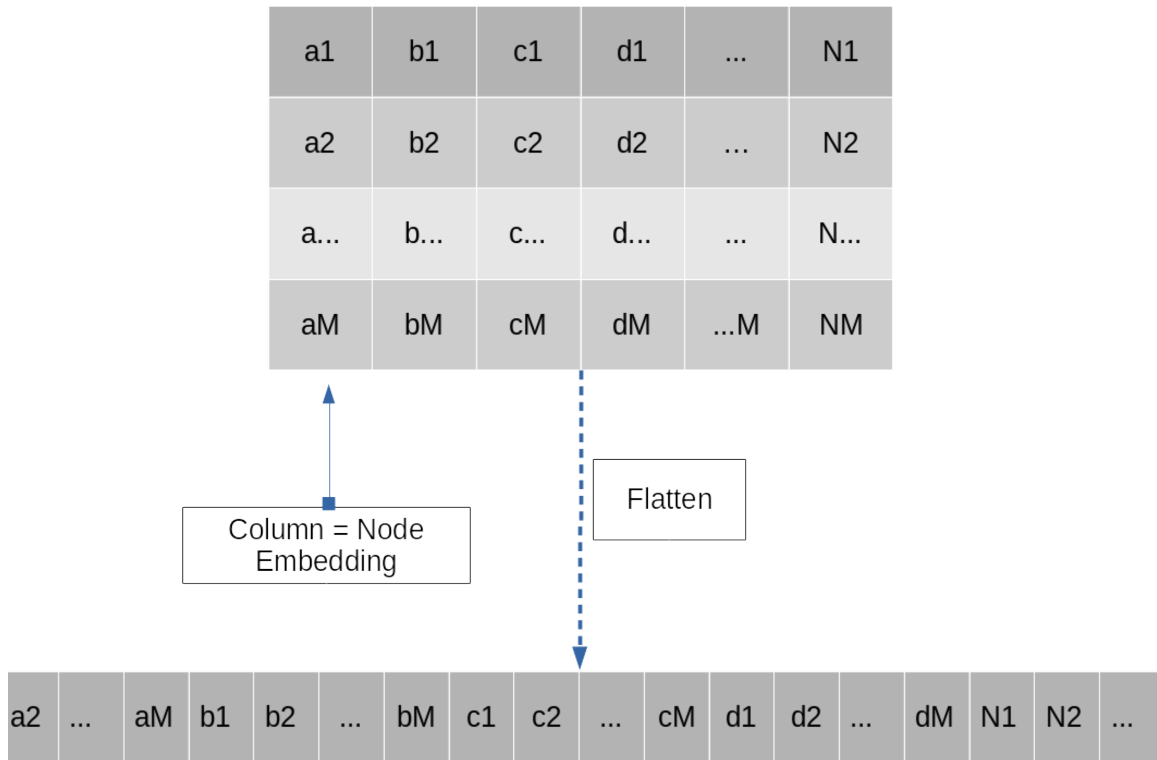


Figure 3.4: Columnwise flattening of N-dimensional graph and M-dimensional embedding allows us to preserve a given nodes embedding in the flattened array allowing an analyst the ability to extract the whole node embedding given one point.

machine learning process. Since the nodes will exist as immediate neighbors in the single dimensional array, we can do simple math, using the length of all embeddings plus their three dimensional size, to extract the individual node's embedding from the array. The ability to do so is important for future work as we can then recreate the generic graph structure using the embeddings and potentially distinguish structures that may be used for further classification.

### 3.0.5 Machine Learning

This research is, in effect, a unique effort in the realm of utilizing graph embeddings for the task of classification on variable length graphs. Thus, many of the methods are used purely as exploratory efforts to distinguish valuable results from pure noise. As mentioned above, we use our dataset of real-world binaries taken from Unix repositories in order to train our models. For training and testing sets, we perform 10-fold cross-validation on our dataset, allowing us reasonable results from our models.

Our current methods for modeling our dataset focus on decision tree classification algorithms. This is an important part of the process as we would like to be able to distinguish the specific extracted information that may represent a specific compiler provenance feature. Decision tree classifiers allow us to see what features may be used to manually extract the features, further allowing us to verify and develop even more features. One such purpose of this is to extract the graph embedding of a node, identify it in the embedding graph, derive the original graph from the embedding, and then identify the structure which the node belong to. Our current approach only applies simple classification methods, future work may use alternative classification methods.

# 4 Results

### 4.0.1 Data Generation

There has been little investigation into Compiler Provenance and as such, there exist no datasets that fit our needs. Datasets are an important part of applying machine learning to a problem as any solution you come to will be a result of the data. As we intend for this method to work in the real world it is important that the data we use is representative. Thus, it was crucial for us to develop a dataset that was easily reproducible and included large numbers of binaries for analysis. For this task, Gentoo, a distribution of Linux, was used in order to automate the binary generation. Unlike other Linux distributions, Gentoo builds all packages from source at installation time. Gentoo allows us to specify a compiler and optimization flags for package building, allowing us to build a large number of binaries with different compilers and flags. Gentoo enables one to generate large binaries that are more representative of applications you would find in the real world. Our final dataset is comprised of roughly 200 programs compiled using the various versions, optimizations, and compilers resulting in a total of 1,700 binaries.

```
COPY --from=gentoo/portage:latest /usr/portage /usr/portage
RUN emerge --autounmask-write -v =sys-devel/gcc-5* || true
RUN etc-update --automode -5
RUN echo CFLAGS=\"-march=native -O2 -pipe -mno-avx\" > /etc/portage/make.conf
RUN FEATURES='-sandbox -userdsandbox' emerge --autounmask-write =sys-devel/gcc-5*
ENTRYPOINT ["/bin/bash", "/tmp/gather.sh", "gcc-5.4.0", "g++-5.4.0"]
CMD []
```

Figure 4.1: Dockerfile
The instructions above are the full dockerfile that is used for generating our binaries for analysis.

```
CC=gcc
CXX=g++
CFLAGS="-march=native -O2 -pipe"
CXXFLAGS="${CFLAGS}"
CHOST="x86_64-pc-linux-gnu"
MAKEOPTS="-j5"
ABI_X86="32 64"

CPU_FLAGS_X86="aes avx avx2 fma3 mmx mmxext popcnt sse sse2 sse3 sse4_1 sse4_2 ssse3"
USE="mmx mmxext sse sse2 ssse3 sse4_1 sse4_2 avx 3dnow \
apng jpeg gif tiff png svg xmp \
flac mp3 ogg opus vorbis pulseaudio \
rtmp webp xvid x264 vaapi glamor \
icu corefonts truetype cjk \
cairo icu minizip opengl xcb libinput qml wayland xwayland gles qt5 \
cryptsetup gcrypt gudev -kdbus kernel-builtin lz4 iproute2 gpg systemd \
contrib extensions nsplugin bash-completion secure-delete bluetooth \
-firmware-loader -tcpd -webkit"

GENTOO_MIRRORS="http://gentoo.osuosl.org/ http://www.gtlib.gatech.edu/pub/gentoo http://distfiles.gentoo.org/"
...
```

Figure 4.2: make.conf
Gentoo allows for many configurable options. The above highlights a few of the options a user can configure to tailor their operating system to their specifications. To specify the compiler and flags one only need change options: CC, CXX, CFLAGS, CXXFLAGS

| Compiler | Version | Optimization |
|---|---|---|
| *gcc* | 4.9 | O0,O1,O2,O3,Wall |
| *gcc* | 5.4 | O0,O1,O2,O3,Wall |
| *gcc* | 7.0 | O0,O1,O2,O3,Wall |
| *clang* | 4 | O0,O1,O2,O3,Wall |
| *clang* | 5 | O0,O1,O2,O3,Wall |
| *clang* | 6 | O0,O1,O2,O3,Wall |

Table 4.1: The compilers, their versions, and the optimizations applied for the binaries in our project.

### 4.0.2 Evaluation

To evaluate our classification model we approached the problem in two ways: Tiered, Non-Tiered. Tiered classifies data assuming some information is known ahead of time, such as compiler and version information. Non-Tiered is classification assuming no information is known ahead of time. In addition to these approaches, we apply both a Tiered and Non-Tiered approach to the following algorithms Node2Vec and HOPE. HOPE is the factorization based graph embedding algorithm. Node2Vec is the random walk based algorithm, and is used as a direct comparison for HOPE in these approaches. Finally, we apply both a Tiered and Non-Tiered approach alongside an Average Size Padding method which uses the HOPE algorithm and seeks to reduce the embeddings overall size by imposing a hard limit on the embedding length.

| Compiler | Accuracy |
|---|---|
| *gcc* | 80% |
| *clang* | 85% |

Table 4.2: Classification of the Compilers using HOPE Embeddings with a Tiered approach. There is a 50% chance of classifying correctly through random chance; better than 50% indicates that the method is successful.

| Compiler | Version | Accuracy |
|---|---|---|
| *gcc* | 4.9 | 63% |
| *gcc* | 5.4 | 70% |
| *gcc* | 7.0 | 96% |
| *clang* | 4 | 64% |
| *clang* | 5 | 64% |
| *clang* | 6 | 64% |

Table 4.3: Classification of the Version assuming known Compiler using HOPE Embeddings with a Tiered approach. (33% random chance of correct classification.)

| Compiler | Version | O0 | O1 | O2 | O3 | Wall |
|---|---|---|---|---|---|---|
| *gcc* | 4.9 | 24.1% | 80% | 35% | 7% | 56.7% |
| *gcc* | 5.4 | 29% | 44.4% | 80% | 30% | 37% |
| *gcc* | 7.0 | 48.6% | 42.5% | 32.5% | 8% | 53.3% |
| *clang* | 4 | 55.6% | 57.4% | 31.5% | 32.7% | 46.7% |
| *clang* | 5 | 48.1% | 59.3% | 22% | 28.3% | 46.7% |
| *clang* | 6 | 25.9% | 46.3% | 46.3% | 32.1% | 33.3% |

Table 4.4: Classification of the Optimizations assuming known Compiler and Version using HOPE Embeddings with a Tiered approach. (20% random chance of correct classification.)

| Compiler | Version | Accuracy |
|---|---|---|
| *gcc* | 4.9 | 56% |
| *gcc* | 5.4 | 60% |
| *gcc* | 7.0 | 94% |
| *clang* | 4 | 56% |
| *clang* | 5 | 58% |
| *clang* | 6 | 62% |

Table 4.5: Classification of Compiler and Version using HOPE Embeddings with a Non-Tiered approach. (15% random chance of correct classification.)

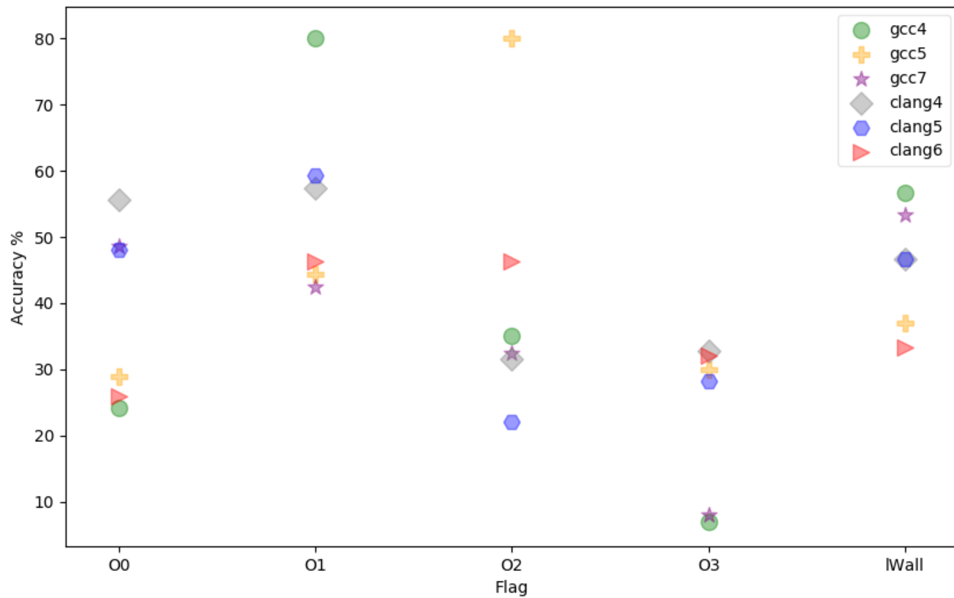


Figure 4.3: This is the visualization of the data shown in Table 4.4. Each column is the optimization applied, each symbol-color combination represents a compiler and version. (20% random chance of correct classification.)

| Compiler | Version | O0 | O1 | O2 | O3 | Wall |
|---|---|---|---|---|---|---|
| *gcc* | 4.9 | 13% | 52% | 24% | 3% | 56.7% |
| *gcc* | 5.4 | 29% | 44.4% | 26% | 23.5% | 26.7% |
| *gcc* | 7.0 | 43% | 37.5% | 17.5% | 16.7% | 36.7% |
| *clang* | 4 | 35% | 37% | 25.9% | 40% | 53.3% |
| *clang* | 5 | 18.5% | 46.3% | 20% | 27% | 30% |
| *clang* | 6 | 16.7% | 37% | 33.3% | 24.5% | 36.7% |

Table 4.6: Classification of Compiler and Version and Optimization using HOPE Embeddings with a Non-Tiered approach. (3% random chance of correct classification.)

| Compiler | Accuracy |
|---|---|
| *gcc* | 55% |
| *clang* | 93% |

Table 4.7: Classification of Compilers using HOPE alongside Average Length Padding in a Tiered approach. (50% random chance of correct classification.)

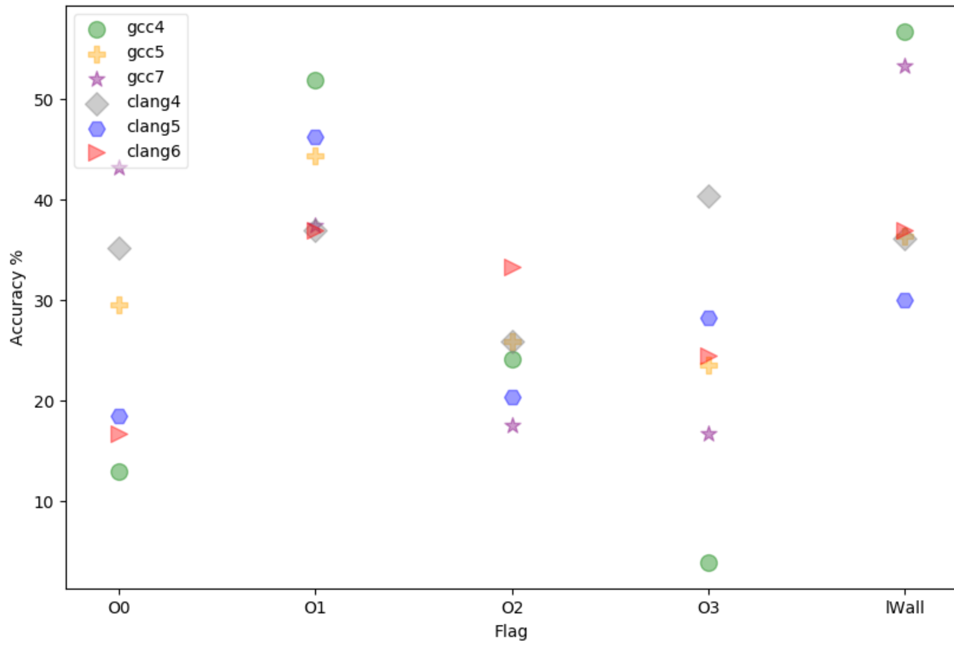


Figure 4.4: Visualization of the data shown in Table 4.6. Each symbol corresponds to a compiler and version and the accuracy predicting the flags used for compilation. (3% random chance of correct classification.)

| Compiler | Version | Accuracy |
|---|---|---|
| *gcc* | 4.9 | 56% |
| *gcc* | 5.4 | 70% |
| *gcc* | 7.0 | 28% |
| *clang* | 4 | 52% |
| *clang* | 5 | 55% |
| *clang* | 6 | 61% |

Table 4.8: Classification of Version knowing Compiler using HOPE Embeddings alongside Average Length Padding with a Tiered approach. (33% random chance of correct classification.)

| Compiler | Version | O0 | O1 | O2 | O3 | Wall |
|---|---|---|---|---|---|---|
| *gcc* | 4.9 | 63% | 25% | 37% | 33% | 5% |
| *gcc* | 5.4 | 43% | 25% | 42% | 13% | 13% |
| *gcc* | 7.0 | 46% | 48% | 37% | 17% | 11% |
| *clang* | 4 | 37% | 20% | 35% | 33% | 40% |
| *clang* | 5 | 27% | 9% | 50% | 26% | 43% |
| *clang* | 6 | 43% | 18% | 43% | 32% | 30% |

Table 4.9: Classification of Compiler, Version and Optimizations using HOPE Embeddings alongside Average Length Padding with a Non-Tiered approach. (3% random chance of correct classification.)

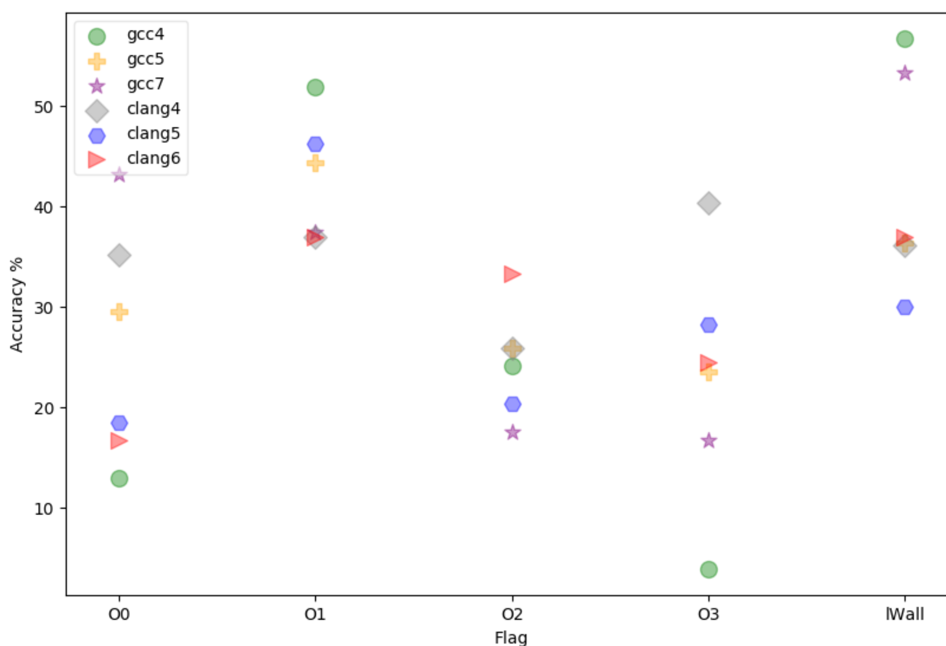


Figure 4.5: Visualization of the data shown in Table 4.9. Each symbol corresponds to a compiler and version and the accuracy predicting the flags used for compilation. (3% random chance of correct classification.)

| Compiler | Version | O0 | O1 | O2 | O3 | Wall |
|---|---|---|---|---|---|---|
| *gcc* | 4.9 | 48% | 35% | 35% | 35% | 0% |
| *gcc* | 5.4 | 23% | 48% | 7% | 36% | 0% |
| *gcc* | 7.0 | 26% | 20% | 35% | 25% | 52% |
| *clang* | 4 | 29% | 38% | 13% | 16% | 0% |
| *clang* | 5 | 30% | 25% | 29% | 33% | 20% |
| *clang* | 6 | 35% | 38% | 20% | 38% | 17% |

Table 4.10: Classification of Compiler, Version and Optimizations using Node2Vec Embeddings with a Non-Tiered approach. (3% random chance of correct classification.)

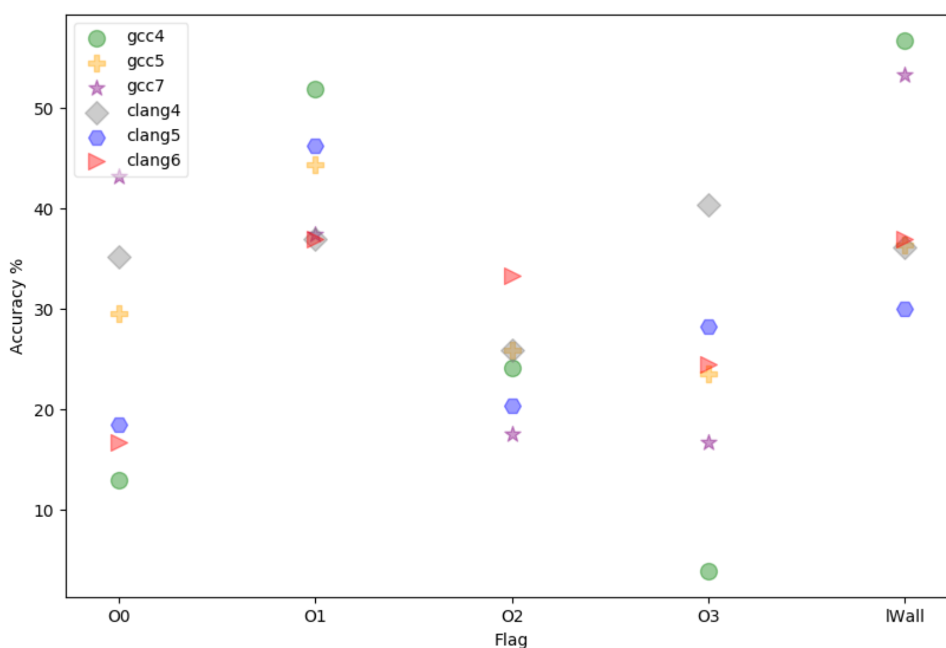


Figure 4.6: Visualization of the data shown in Table 4.10. Each symbol corresponds to a compiler and version and the accuracy predicting the flags used for compilation. (3% random chance of correct classification.)

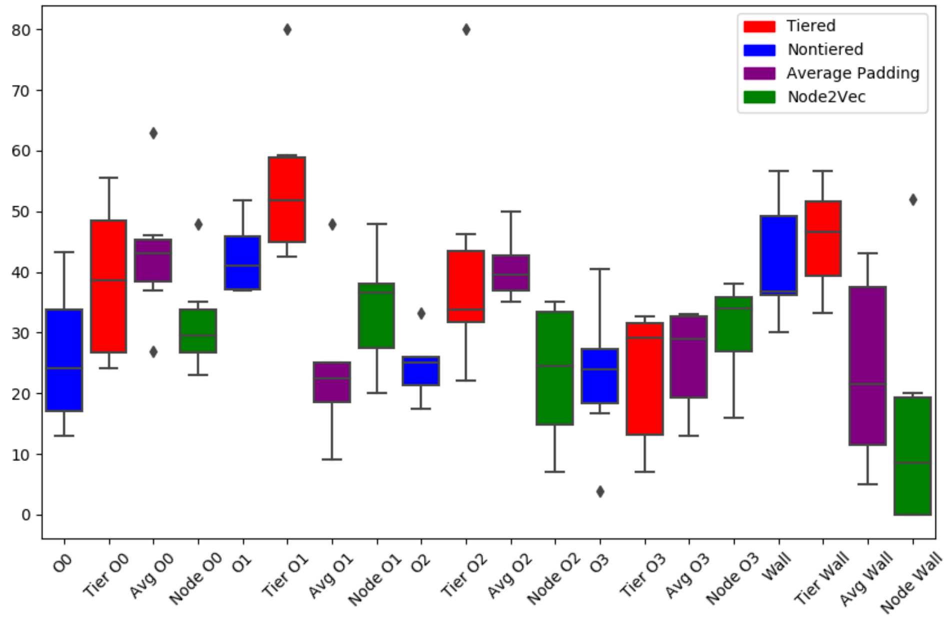


Figure 4.7: Boxplots displaying the accuracy of each approach for each flag. Red shows tiered accuracy. Blue shows the Non-tiered accuracy. Purple highlights the average padding approach accuracy. Green displays the accuracy using Node2Vec embeddings. Diamonds indicate outliers. The random chance of success for Tiered was 20%, while for all other methods it was 3%.

### 4.0.3 Limitations

Our experiments suffer from a few limitations, related largely to the implementation and dataset generation. First, all resulting CFGs stem from the disassembly methods described in our implementation. The current disassembly method used is a simple linear method, which has the potential to miss large segments of binary. Linear disassembly can give us a rough structure of the CFG for a given binary. Unfortunately, as linear disassembly only runs once through the disassembly, it often produces large, disconnected segments of nodes. Thus, any embeddings developed from the disconnected graphs are a reflection of this approach.

Furthermore, our decision to use a linear disassembly method was aided by x86's consistency with realigning itself as machine code is generated. Linear disassembly continuously disassembles any binary data it sees until it encounters invalid binary data. The linear approach may become offset from the intended binary data creating an entirely fictional graph representation of the binary data, however, due to x86s propensity to realign itself we are less likely to generate completely fictional representations.

Lastly, graph embeddings result in multidimensional data, which is made into a one dimensional array in our approach and then padded to make all embeddings an equivalent length for our machine learning method. Making the data into a one-dimensional form is important for our machine learning model as multi-dimensional data is a much larger problem for classification. This padding and flattening loses both data and may insert some bias into our dataset, and it would be worth exploring more in future work.

### 4.0.4 Discussion

Our results indicate that the utilization of graph embeddings for compiler provenance feature extraction has potential. Classification was done using four different approaches: tiered, non-tiered, average padding, Node2Vec. In our tiered approach, the HOPE embedding algorithm was used with the assumption that some information is known beforehand such as the compiler or version. Table 4.2 shows our method distinguishes between our two compilers with relatively high accuracy. Further distinguishing between versions as in Table 4.3 again shows us that the model can distinguish between the versions with some confidence. Lastly, given compiler and version, the model can distinguish between most of the optimization levels with improved accuracy, better than chance, however, optimization level three drops below even random chance (discussed below). For the non-tiered approach, no prior knowledge is assumed. Again, using the HOPE algorithm, all predictions are the same or significantly higher than random chance. Average Padding is an alternative method for dealing with the variable length embeddings. By collecting the embeddings for all binaries and determining the average length we cut off those embeddings which are longer and pad those that are smaller, which resulted in tighter variance for overall accuracy. Finally, Node2Vec was used as a single alternative to the HOPE embedding to present the effectiveness of random walk based graph embedding methods. The random walk method resulted in positive results, however, was much more variable and had subpar accuracy when it came to Wall optimizations.

When comparing the results for each compiler it is important to note the differences between the compilers, versions, and flags used for compilation. Our hypothesis was that the underlying structure of the CFGs for the various compilers was different enough that there was information we could extract to distinguish between them.

The results do indicate that the underlying structure produced by each compiler is different enough to distinguish between the two compilers in the dataset with relatively high confidence. This is understandable, given that compiler, such as clang and gcc are implemented differently and are optimized in different ways. Their implementation in regards to branching structures, symbol tables, libraries, and data structures will all affect the output CFG that we use as the first step in our approach.

Version information becomes slightly more difficult due to the slight optimizations that may have been made from one version to the next. Major version numbers indicate that there were significant changes in the implementation. Our hypothesis is that the new features added to each major version would create new structures that could allow us to distinguish between major versions. Our results support this hypothesis. Table 4.3 shows that the accuracy for, gcc with the tiered approach, is much higher than clang. This larger variation in version accuracy for gcc may be partly related to the version jump from 5.4 to 7.0, which may contain more features and give a better detection for gcc 7.0.

Finally, distinguishing between flags used for compilation can be extremely difficult, due to the effects they have on the resulting machine code. Some flags such as the debug flag result in no distinguishable difference during CFG generation, making it impossible to extract. Optimization flags are thus the most important flags to focus on due to their underlying effects of the machine code output. Each optimization level results in different translations being made from source to machine code, with optimization level zero ("O0") resulting in no translation taking place. Often optimization level two ("O2") is used for packages with optimization level three ("O3") rarely being used due to it increasing the size of the binary and, potentially, its runtime. "Wall" is not considered an optimization flag however, it enables many features such as bounds checking to add more logic to be applied during the source to machine

code translation. Results from Tables (4.4, 4.6, 4.10, 4.9) indicate that these flags impart some additional graph features that this approach can use for compiler provenance detection. Comparing tiered vs non-tiered approaches enabled us to investigate which features our model may be more able to detect and extract.

Tiered and non-tiered approaches are noticeably ineffective at predicting the “O3” optimization level. Optimizations applied at this level include all optimizations done by “O2”, are generally more expensive, and include other techniques, such as: function inlining, loop unrolling, and vectorization. Function inlining replaces the called function with the body of the called function, eliminating the overhead of calling a new function. Loops are conditional repeated operations such as a for loop and are normally represented as three separate blocks in a CFG: one for the condition, one for the operations to be done, one for exit. Loop unrolling removes the conditional portion of the loop and unpacks the loop into one block, which reduces the number of features that may be represented in the CFG. Vectorization takes advantage of compiler specific functions that are separate from user or library code, which make certain operations much faster. These optimizations can have a significant effect on a program’s machine code and the time it takes for a binary to be generated. It is possible that due to our simple linear disassembly we miss potentially vectorized operations, because vectorization requires that disassembly occurs in sections of memory not included in our initial linear sweep. Additionally, our method might miss some of the inlined code due to our simple disassembly method. In a completely perfect CFG one would see interconnecting branches from function call to the functions interconnecting nodes, however with our approach if a function is called the node may not show a connection. Thus it is completely possible to miss function inlining without a more effective disassembly method. Furthermore, as optimizations in “O3” may be excluded during compilation due to errors, memory issues, or other factors, the

resulting machine code may be extremely similar to optimization level “O2” causing low prediction accuracy..

Conversely, Node2Vec and Average Length Padding approaches seem to struggle less with optimization level “O3” while completely missing, as in Node2Vec, accuracy for optimization level “Wall”. The “Wall” flag is meant to add bounds checking and other security measures that aren’t included by default during compilation. Node2Vec’s random walk approach can be ideal for networks with many neighbors, however, the effect of this approach on a directed graph such as the developed CFGs may be ineffective due to the extremely low amount of neighbors each node may have. Most notably, as bounds checking is implemented by “Wall” optimization flag, the resulting CFG may simply may not be that different from the other optimization levels if there is no bounds checking required by the original source code. The biased random walk that is used with Node2Vec may develop embeddings that do not accurately portray the CFGs in which bounds checking is implemented, this could result in highly variant predictions as seen in Table 4.10.

Average Length Padding suffers from the same missing embedding issue that Node2Vec suffers from, due to the cutoff that may be applied to each embedding. This often results in large removed graph segments across all embeddings. Overall, Average Length Padding may be more useful if applied along with a preprocessing step to select potential nodes in a graph as an alternative to using it as a cutoff due to missing data. Node2Vec is something that may be more useful given potentially larger CFGs with more information contained in each node.

# 5 Conclusions

### 5.0.1 Future Works

This work highlights the novel use of graph embeddings for binary analysis and compiler provenance detection. Future work may focus on disassembly, applying more advanced linear, predictive, and recursive techniques on binaries. Further analysis of graph embedding techniques for their applicability in the field of binary analysis is another step for future work. There are many graph embedding techniques, and looking for those that scale well and other features. Finally reconstructing the graphical structure from the important features extracted from our approach. A key feature of the GEM framework was analysing their reproduction of the graphical structure after embedding (Goyal and Ferrara 2017). This reproduction would be the next logical step after feature extraction, recreating the structures that are represented in the embedding and are thus indicative of the compiler provenance features.

CFG generation is a pivotal step in our approach for compiler provenance detection. All data is derived from this step and requires that it accurately represents the application itself. Future work may focus on implementing a more verbose and effective approach for CFG generation. The current approach outlined in this paper uses a linear sweep for disassembly in order to generate the CFGs, however, as mentioned above a linear sweep suffers from offset issues and missing segments of machine code. Further complicating the process is that using a recursive approach can suffer from obfuscation and inefficient disassembly times. Developing a hybrid approach

that can accurately disassemble a given binary without inferring anything from the applications metadata is a critical future step.

Along with CFG generation is the issue of embeddings, padding, and classification. Embeddings themselves are a result of the graph that is being embedded, as such every embedding will be of different sizes given the graph it is working with. In our real world setting we operate assuming that all binaries are different, thus the necessity for padding embeddings to equal length as outlined above. However, future work may focus on a few aspects of this problem. First, the issue of dissimilar embeddings. A potential avenue for future research may develop an approach that extracts similar structures within a CFG such that the resultant embedding is of similar length to others which may then be used for classification. Second, the issue of padding. Currently our approach assumes the padding does not impact the embedding in any way, future work may look at other forms of padding such as: adding nodes to the graph prior to embedding and different values for embedding. Last, the classification of these embeddings. As machine learning requires that all labels be consistent across samples. Our approach made the assumption that the labels would remain consistent for each. Future work may focus on developing a scheme such that the resultant embeddings remain provably consistent when doing classification.

Finally utilization of embeddings for classification require much more research. Our current approach discusses embeddings for use only in relation to simple decision tree and random forest classification. Both these are relatively simple methods of classification and leave much to be desired in regards to efficacy and optimization. Future work may look at using neural networks and deep learning methods, which are much more effective with variable length input, for compiler provinence idenfication and classification.

### 5.0.2 Conclusion

In this paper, we presented a novel technique for detecting and recovering the compiler provenance features of a binary using simple disassembly techniques coupled with graphical embeddings and machine learning. This approach is novel in its use of graph embeddings for machine learning, allowing us to highlight its applicability in the field of reverse engineering. Our results show that these features show promise in their ability to help distinguish between compiler versions, flags, and optimization levels when applied in a tiered or non-tiered manner. Furthermore, our results indicate that alternative graph embeddings also have the potential for use in compiler provenance identification. Most importantly our results show that with more refinement and resources this could be useful in the field of reverse engineering and binary analysis.

# References

Behera, Chandan Kumar and D. Lalitha Bhaskari (2015). "Different Obfuscation Techniques for Code Protection". In: *Procedia Computer Science* 70. Proceedings of the 4th International Conference on Eco-friendly Computing and Communication Systems, pp. 757–763. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2015.10.114. URL: http://www.sciencedirect.com/science/article/pii/S1877050915032780 (cit. on p. 8).

Bonfante, Guillaume, Matthieu Kaczmarek, and Jean-Yves Marion (2007). "Control Flow Graphs as Malware Signatures". In: (cit. on p. 11).

Gotlieb, Arnaud, Bernard Botella, and Michel Rueher (1998). "Automatic Test Data Generation Using Constraint Solving Techniques". In: *ISSTA* (cit. on p. 11).

Goyal, Palash and Emilio Ferrara (2017). "Graph Embedding Techniques, Applications, and Performance: A Survey". In: *CoRR* abs/1705.02801. arXiv: 1705.02801. URL: http://arxiv.org/abs/1705.02801 (cit. on pp. 11, 12, 18, 36).

Grover, Aditya and Jure Leskovec (2016). "node2vec: Scalable feature learning for networks". In: *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, pp. 855–864 (cit. on p. 13).

Hex-Rays (2019). URL: https://www.hex-rays.com/products/ida/ (cit. on pp. 6, 11).

Hosseinzadeh, Shohreh et al. (2018). "Diversification and obfuscation techniques for software security: A systematic literature review". In: *Information and Software Technology* 104, pp. 72–93. ISSN: 0950-5849. DOI: https://doi.org/10.1016/j.infsof.2018.07.007. URL: http://www.sciencedirect.com/science/article/pii/S0950584918301484 (cit. on p. 7).

ioccc (2019). URL: https://www.ioccc.org/ (cit. on p. 7).

Jacobson, Emily R., Nathan Rosenblum, and Barton P. Miller (2011). "Labeling Library Functions in Stripped Binaries". In: *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*. PASTE '11. Szeged, Hungary: ACM, pp. 1–8. ISBN: 978-1-4503-0849-6. DOI: 10.1145/2024569.2024571. URL: http://doi.acm.org/10.1145/2024569.2024571 (cit. on pp. 4, 5).

Krügel, Christopher et al. (2004). "Static Disassembly of Obfuscated Binaries". In: *USENIX Security Symposium* (cit. on pp. 4, 6, 7).

Meng, Xiaozhu and Barton P. Miller (2016). "Binary Code is Not Easy". In: *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ISSTA 2016. Saarbrücken, Germany: ACM, pp. 24–35. ISBN: 978-1-4503-4390-9. DOI: 10.1145/2931037.2931047. URL: http://doi.acm.org/10.1145/2931037.2931047 (cit. on p. 6).

Mikhailov, A. et al. (2016). "Control flow graph visualization in compiled software engineering". In: *2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pp. 1313–1317. DOI: 10.1109/MIPRO.2016.7522343 (cit. on p. 11).

NSA (2019). URL: https://www.nsa.gov/resources/everyone/ghidra/ (cit. on p. 6).

Ou, Mingdong et al. (2016). "Asymmetric Transitivity Preserving Graph Embedding". In: *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '16. San Francisco, California, USA: ACM, pp. 1105–1114. ISBN: 978-1-4503-4232-2. DOI: 10.1145/2939672.2939751. URL: http://doi.acm.org/10.1145/2939672.2939751 (cit. on pp. 11, 12, 18).

Popov, Igor V., Saumya K. Debray, and Gregory R. Andrews (2007). "Binary Obfuscation Using Signals". In: *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*. SS'07. Boston, MA: USENIX Association, 19:1–19:16. ISBN: 111-333-5555-77-9. URL: http://dl.acm.org/citation.cfm?id=1362903.1362922 (cit. on pp. 7, 8).

Rahimian, Ashkan et al. (2015). "BinComp: A stratified approach to compiler provenance Attribution". In: *Digital Investigation* 14. The Proceedings of the Fifteenth Annual DFRWS Conference, S146–S155. ISSN: 1742-2876. DOI: https://doi.org/10.1016/j.diin.2015.05.015. URL: http://www.sciencedirect.com/science/article/pii/S1742287615000602 (cit. on pp. 2, 3, 5).

Rosenblum, Nathan E., Barton P. Miller, and Xiaojin Zhu (n.d.). *Extracting Compiler Provenance from Program Binaries* (cit. on p. 5).

Schwarz, B., S. Debray, and G. Andrews (2002). "Disassembly of executable code revisited". In: *Ninth Working Conference on Reverse Engineering, 2002. Proceedings.* Pp. 45–54. DOI: 10.1109/WCRE.2002.1173063 (cit. on p. 7).

Taylor, Sean (2017). URL: https://www.defcon.org/images/defcon-17/dc-17-presentations/defcon-17-sean_taylor-binary_obfuscation.pdf (cit. on p. 9).

Vector-35 (2019). URL: https://binary.ninja/ (cit. on p. 6).

Wang, Ruoyu et al. (2017). "Ramblr: Making Reassembly Great Again". In: *NDSS* (cit. on pp. 2, 6).

Wang, S., P. Wang, and D. Wu (2016). "UROBOROS: Instrumenting Stripped Binaries with Static Reassembling". In: *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. Vol. 1, pp. 236–247. DOI: 10.1109/SANER.2016.106 (cit. on p. 2).

Xu, Liang, Fangqi Sun, and Zhendong Su (2010). "Constructing Precise Control Flow Graphs from Binaries". In: (cit. on p. 11).