Binary Code Fingerprinting with Application to Automated Vulnerability Detection

Paria Shirani

A Thesis
in
The Concordia Institute
for
Information Systems Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of
Doctor of Philosophy (Information and Systems Engineering) at
Concordia University
Montréal, Québec, Canada

February 2021

© Paria Shirani, 2021

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: **Paria Shirani**

Entitled: **Binary Code Fingerprinting with Application to Automated Vulnerability Detection**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Information and Systems Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

______________________ Chair
Dr. Emad Shihab

______________________ External Examiner
Dr. Dali Kaafar

______________________ External to Program
Dr. Juergen Rilling

______________________ Examiner
Dr. Amr Youssef

______________________ Examiner
Dr. Mohammad Mannan

______________________ Thesis Supervisor
Dr. Mourad Debbabi

______________________ Thesis Supervisor
Dr. Lingyu Wang

Approved by ______________________
Dr. Mohammad Mannan, Graduate Program Director

January 21, 2021 ______________________
Dr. Mourad Debbabi, Dean
Gina Cody School of Engineering and Computer Science

Abstract

Binary Code Fingerprinting with Application to Automated Vulnerability Detection

Paria Shirani, Ph.D.

Concordia University, 2021

With the growing popularity of emerging technologies, the prevalence of digital systems is more than ever. Security, however, has still lagged behind, as evidenced by the increasing rate of recent attacks. Oftentimes, cyber-attacks are initiated by running a malicious code or by exploiting vulnerabilities in the underlying software. To mitigate such alarming threats, analyzing software binary code (a.k.a. *binary analysis*) has been known as a promising solution. This thesis answers the following research question: *how to automatically fingerprint a cross-architecture code with optimization and obfuscation by attributing compiler provenance, identifying library functions, and detecting vulnerable functions?* Specifically, it first analyzes the syntax, structure, and semantic of functions to extract compiler provenance in cross-complied binaries. Second, it introduces a single robust function signature based on heterogeneous features to solve library function identification problem. Third, it overcomes vulnerable function detection problem through a multi-stage fuzzy matching approach on firmware images. Finally, it addresses vulnerability detection problem in cross-architecture obfuscated binaries and firmware images through a neural machine translation-based approach. This thesis advances the state-of-the-art by improving the accuracy, scalability, and efficiency of binary code analysis. All of the proposed approaches are implemented as a prototype system and their performance are evaluated with real data.

Dedication

This thesis is dedicated with love and affection to:
My Beloved FAMILY*,*
My MOTHER*, a beautiful, kind, positive, and gentle soul, who gave me many life lessons, and I wish I could have had her longer in my life;*
My SON*, who is indeed a treasure and gift that I receive every day, and his presence gave me motivation, hope and positive energy to continue;*
My HUSBAND*, who is also my best friend and a great leader, the first one who directed and encouraged me to start this path and always provided the best to keep me happy.*

Acknowledgments

I would like to express my sincere gratitude and appreciation to all those who have made this work possible. Foremost, I would like to express my heartfelt appreciation to Dr. Mourad Debbabi and Dr. Lingyu Wang, who have kindly accepted to be my supervisors and highly contributed to the supervision and fulfillment of this thesis. The invaluable leadership and support of Dr. Debbabi, his great vision and insights, his constructive guidance besides his kindness and charisma, significantly contributed to the success of this life-changing journey. Indeed, his outstanding contribution is beyond the success of a thesis; it highly contributes to the growth of a person, a researcher and consequently a society. I am also profoundly grateful to Dr. Wang who significantly changed the direction of my life, and kindly provided me with invaluable comments, suggestions, guidance, and encouragement, without which this would not happen.

My gratitude extends to Dr. Dali Kaafar for accepting to serve as the external examiner and for his time and precious contribution to the evaluation of this thesis. I am also thankful to Dr. Juergen Rilling, Dr. Mohammad Mannan, and Dr. Amr Youssef for serving as my supervisory committees and providing valuable feedback and insightful suggestions throughout this journey.

I would also like to thank Dr. Aiman Hanna and Dr. Serguei A. Mokhov for their continuous and valuable support throughout this journey. I am very thankful to Dr. Saed Alrabaee, who generously and continuously guided me through this process when everything was new to me. I am also very grateful to Dr. Suryadipta Majumdar and Dr. Andrei

Tudor Soeanu Caval for their selfless support, especially through hard times. Further thanks to other lab-mates and friends in the Security Research Centre (SRC), Dr. ElMouatez Billah Karbab, Dr. Amine Boukhtouta, Leo Chollard, Ling Tan, He Huang, Anthony Andreoli, Dr. Djedjiga Mouheb, Ashkan Rahimian, Aniss Chohra, Dr. Bassam Moussa, Dr. Sujoy Ray, Mina Khalili, Lina Nouh, Pratyusha Bhattacharya, Quentin Le Sceller, Perry Jones, Houssem Eddine Bordjiba, Aref Mourtada, Badis Racherache and Dhiaa Rebbah for their support and memories shared in the laboratory.

I owe special thanks to Quebéc/Canada for accepting me as a new immigrant and the Concordia Institute for Information Systems Engineering (CIISE) for accepting me as a PhD student. I also acknowledge the financial support of Natural Sciences and Engineering Research Council (NSERC), Fonds de la recherche en santé du Québec (FRQNT), Google Canada, Hydro-Québec, Thales Canada, Arbour Foundation, Defence Research and Development Canada (DRDC), and Concordia University without which this entire degree would not have been possible.

Last but not least, I would like to express my deepest gratitude to my beloved family who sacrificed throughout this long journey and provided me with endless moral support, immense comfort, encouragement and love. They always inspired me to be a strong and better person.

THANK YOU ALL!

Contents

List of Figures xiii

List of Tables xvi

1 Introduction 1

1.1 Motivations . . . 1

1.2 Problem Statement . . . 6

1.2.1 Research Questions . . . 6

1.2.2 Threat Model . . . 8

1.3 Research Contributions . . . 9

1.3.1 Survey on Static Binary Analysis Approaches . . . 9

1.3.2 Compiler Provenance Attribution . . . 10

1.3.3 Library Function Identification . . . 11

1.3.4 Vulnerability Detection in Firmware Images . . . 11

1.3.5 Cross-Architecture Code Similarity and Vulnerability Detection . . . 12

1.3.6 Summary . . . 13

1.4 Thesis Organization . . . 14

2 Literature Review 16

2.1 Binary Analysis Overview and Challenges . . . 16

2.1.1 Overview . . . 17

2.1.2 Challenges . . . 18
2.2 Preliminaries for Static Approaches . . . 22
2.2.1 Intermediate Representation . . . 23
2.2.2 Feature Extraction . . . 24
2.2.3 Application Domains . . . 27
2.3 Static Binary Analysis Approaches . . . 32
2.3.1 Graph-Based Approaches . . . 33
2.3.2 Data-Flow-Based Approaches . . . 37
2.3.3 Distance-Based Approaches . . . 41
2.3.4 Symbolic Execution Combined with Static Analysis . . . 44
2.3.5 Comparative Study . . . 45
2.4 Summary . . . 51

3 Compiler Provenance Attribution 52
3.1 Introduction . . . 53
3.1.1 Motivating Example . . . 55
3.1.2 Approach Overview . . . 59
3.2 Feature Extraction . . . 61
3.2.1 Compiler Transformation Profile . . . 61
3.2.2 Compiler Tags . . . 64
3.2.3 Compiler Functions . . . 64
3.2.4 Annotated Control Flow Graph . . . 67
3.2.5 Compiler Constructor and Terminator . . . 70
3.3 Multi-layered Compiler Provenance Attribution . . . 71
3.3.1 Layer 1: Compiler Family Identification . . . 72
3.3.2 Layer 2: Compiler Function Labelling . . . 74
3.3.3 Layer 3: Version and Optimization Recognition . . . 78

3.4 State-of-the-Art Compiler Provenance Extraction . . . 79
3.4.1 Overview . . . 79
3.4.2 Dataset Generation . . . 80
3.4.3 Evaluation Results . . . 80
3.4.4 Discussion . . . 83
3.5 Evaluation . . . 83
3.5.1 Dataset Preparation . . . 83
3.5.2 Accuracy Results . . . 84
3.5.3 Comparison . . . 87
3.6 Limitations and Concluding Remarks . . . 87

4 Library Function Identification **89**
4.1 Introduction . . . 90
4.1.1 Motivating Example . . . 92
4.1.2 Approach Overview . . . 93
4.2 Feature Extraction . . . 95
4.2.1 Graph Feature Metrics . . . 95
4.2.2 Instruction-level Features . . . 97
4.2.3 Statistical Features . . . 98
4.2.4 Function-Call Graph . . . 99
4.2.5 Feature Selection . . . 100
4.3 Detection . . . 102
4.3.1 B^{++}tree Data Structure . . . 102
4.3.2 Filtering . . . 104
4.4 Evaluation . . . 106
4.4.1 Experimental Setup . . . 106
4.4.2 Dataset Preparation . . . 107

4.4.3 Function Identification Accuracy Results 108
4.4.4 Impact of Obfuscation 109
4.4.5 Impact of Compilers 112
4.4.6 Impact of Feature Selection 112
4.4.7 Impact of Filtering 114
4.4.8 Scalability Study 114
4.4.9 Application to Malware 116
4.5 Limitations and Concluding Remarks 117

5 Vulnerable Library Function Detection in Binaries and Firmware 118
5.1 Introduction 119
5.2 Approach Overview 122
5.3 Building IED Firmware and Vulnerability Databases 125
5.3.1 Intelligent Electronic Devices in the Smart Grid 126
5.3.2 Manufacturer Identification 128
5.3.3 Vulnerability Database 129
5.3.4 Firmware Database 131
5.4 Multi-stage Detection Engine 133
5.4.1 Function Shape-Based Detection 134
5.4.2 Path-Based Detection 141
5.4.3 Fuzzy Matching-Based Detection 147
5.5 Evaluation 151
5.5.1 Experimental Setup 151
5.5.2 Library Function Identification Accuracy 152
5.5.3 Efficiency 153
5.5.4 Comparison 154
5.5.5 Detecting Vulnerabilities in Real Firmware 156

5.5.6 Impact of Multiple Detection Stages 158
5.5.7 Impact of Parameters 159
5.5.8 Scalability Study 159
5.6 Limitations and Concluding Remarks 160

6 Code Similarity Detection in Cross-Architecture Obfuscated Binaries 161
6.1 Introduction 162
6.2 Approach Overview 165
6.3 Vulnerability Database Generation 166
6.4 Function Representation Generation 169
6.4.1 Intermediate Representation 169
6.4.2 Feature Engineering 172
6.5 Code Similarity Detection 177
6.5.1 LSTM Encoder-Decoder 177
6.5.2 Input Pre-Processing 179
6.5.3 Function Matching 183
6.6 Evaluation 184
6.6.1 Experimental Settings 184
6.6.2 Comparison 185
6.6.3 Accuracy Results 191
6.6.4 Hyper-parameters Selection 194
6.6.5 Out-Of-Vocabulary Words 195
6.6.6 Efficiency 197
6.6.7 Detecting Vulnerabilities on Real-World Firmware 199
6.7 Limitations and Concluding Remarks 200

7 Conclusion 202

Bibliography **207**

List of Figures

2.1 Taxonomy of features 25

2.2 Taxonomy of application domains 28

2.3 Taxonomy of static-based binary analysis approaches 32

3.1 BINCOMP multi-layered architecture 60

3.2 Code transformation example 63

3.3 ACFG construction of __do_global_ctors function 70

3.4 Sample of *GCC* compiler constructor and terminator 71

3.5 Supervised compilation steps and produced artifacts 72

3.6 Intersecting programs to obtain common compiler functions 75

3.7 ECP accuracy results 81

3.8 Impact of the threshold value 81

3.9 Diversity in datasets 82

3.10 Feature extraction and feature ranking time 82

3.11 The accuracy results against different compilers 85

3.12 Accuracy comparison of BINCOMP, ECP, and IDA PRO 87

4.1 Control flow graphs of _memmove, _memchr, and _lock_file 93

4.2 BINSHAPE approach overview 94

4.3 Indexing and detection structure 104

4.4 ROC curve for BINSHAPE features 109

4.5 CFGs of EC_EX_DATA_free_all_data function in OpenSSL - libssl 111

4.6 Impact of top-ranked features 113
4.7 Impact of best features 114
4.8 Impact of filtering 115
4.9 Scalability study 115
5.1 BINARM overview 122
5.2 Comparing the CFGs of a vulnerable function and an unknown function . . 125
5.3 Smart grid overview 126
5.4 Four applications in the smart grid and some examples of related IEDs . . . 127
5.5 Distribution of hardware architectures amongst collected IED firmware . . 129
5.6 BINARM: multi-stage detection engine 133
5.7 Mutual information of features 138
5.8 An example of two *weighted paths* of $w1$ and $v1$ 142
5.9 Mutual information of mnemonics 144
5.10 Weight assignment example 146
5.11 Fuzzy matching 149
5.12 CDF of vulnerable function search time 154
5.13 A snapshot of BINARM's in-depth results for vulnerability search 157
5.14 Impact of parameters 159
5.15 Scalability study 160
6.1 TIOHTIÀ:KE approach overview 165
6.2 CFGs of `zlib-inflateMark` function in x86 and ARM architectures . . 170
6.3 VEX-IR of `zlib`-v1.2.8-`inflateMark` function 171
6.4 First VEX-IR basic block in `zlib`-v1.2.8-`inflateMark` function 172
6.5 CFGs of `libgmp-default-allocate` function in x86 architecture . . 174
6.6 CFGs of `libgmp-default-allocate` function in ARM architecture . 174
6.7 TIOHTIÀ:KE code similarity solution 178

6.8 Randomly selected functions in ImageMagick v7.0.6.10 library 181
6.9 Function summarizing 182
6.10 Empirical distribution in binaries compiled with different obfuscations . . . 188
6.11 Empirical distribution in binaries compiled with all optimizations 190
6.12 Empirical distribution in two different binaries 192
6.13 Empirical distribution in binaries compiled with different compilers 193
6.14 Effects of epoch on AUC and loss 194
6.15 Evaluation on out-of-vocabulary tokens 196
6.16 Efficiency evaluation 197
6.17 Training time with varying embedding size and latent dimension 198
6.18 Training time with varying embedding size and number of network layers . 199

List of Tables

2.1 A comparison of state-of-the-art static approaches 46

2.2 A comparison of static analysis implementations and evaluations 47

3.1 Proposed mnemonic and operand types grouping 64

3.2 Symbolic features 65

3.3 Numerical and symbolic features 66

3.4 Instruction patterns for annotation 68

3.5 Different compilers and compilation settings used to build the dataset . . . 84

3.6 F-measure results 84

3.7 Accuracy results for variations of compiler versions 86

3.8 Accuracy for variations of compiler optimization levels 86

4.1 Examples of graph metrics 96

4.2 Comparing graph features of _memcpy_s and _strcpy_s 96

4.3 Example of instruction-level features 97

4.4 Example of mnemonic groups 98

4.5 An excerpt of the projects included in our dataset 107

4.6 F-measure before and after applying obfuscation 111

4.7 Impact of compilers and optimization settings 113

4.8 Function identification in Zeus 116

5.1 Identified major smart grid manufacturers and their supported components . 128

5.2 Top-25 vulnerable open-source libraries in identified manufacturers 130

5.3 Function shape features 135
5.4 Instruction normalization 136
5.5 General register normalization 137
5.6 Proposed menmonic groups for ARM instruction set 143
5.7 Accuracy results for library function detection 153
5.8 Baseline comparison on indexing time (in minutes) of `ReadyNAS` 155
5.9 Baseline comparison on search time (seconds) per function 156
5.10 Identifying CVEs in real-world firmware images 157
5.11 Impact of detection stages 158
6.1 Top-25 identified vulnerable open-source libraries used in smart grid IEDs . 167
6.3 Proposed normalization for IR *expressions* 170
6.4 Proposed normalization for IR *statements* 170
6.5 Accuracy results between the original and obfuscated binaries 187
6.6 Accuracy results between different compiler optimizations 189
6.7 Accuracy results of TIOHTIÀ:KE using Precision 191
6.8 Accuracy results of TIOHTIÀ:KE using Recall 191
6.9 Impact of different hyper-parameters 195
6.10 Identifying CVEs in real-world firmware images 199

Chapter 1

Introduction

In this chapter, we first discuss the motivation of this thesis, then define its problem statement, and finally present its main research contributions.

1.1 Motivations

In recent years, given the wide-spread adoption of emerging technologies (e.g., Internet of Things (IoT) and cloud computing), the reliance on digital systems and information technology has been significantly increased. However, this growing popularity of digital systems also turns them into a popular subject of cybersecurity threats. On the other hand, the number of vulnerabilities are increasing[1] attackers exploit the security vulnerabilities in existing software or deliberately develop various malicious software (a.k.a. malware) to compromise those systems. In addition, the increasing complexity and diversity introduced by those technologies in a system multiplies the possibility of design and implementation flaws in those systems. This may lead to various security vulnerabilities. In particular, most

[1] https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cvss-severity-distribution-over-time. Accessed on Jan 21, 2021.; for instance, $17,447$ vulnerabilities are recorded in 2020, which is the fourth consecutive year with a record number of security flaws published. As such,

security vulnerabilities are resulting from the flaws in their underlying software/firmware code[2]. For instance, the presence of software vulnerabilities is one of the most common causes of data breaches[3], which resulted into the average cost of $4.50 million as of 2020. Moreover, state-backed IoT malware show that targeted attacks on IoT devices can evade traditional cybersecurity detection and cause catastrophic failures with significant impact to critical infrastructure. Examples include *Industroyer* (also referred to as *CRASHOVERRIDE*) [48, 196] targetting Ukraine's power grid to control substation switches and circuit breakers, and *BlackEnergy* [164] against the Ukranian's train railway and electricity generation utilities. In addition, according to the study conducted by Lloyd's and the University of Cambridge's Centre for Risk Studies [79], a large-scale cyberattack can lead to a $243 billion to $1 trillion loss to the U.S. economy.

Consequently, analyzing software systems based on their binary code (namely, *binary analysis and code fingerprinting*), especially where the source code is unavailable (e.g., malware or closed-source software), is an absolute necessity. Moreover, binary analysis has a larger scope (e.g., the whole application including the libraries) than source code analysis, in which is limited to the visible source code and may not cover the vulnerabilities in the compiled code. Furthermore, compilers and tool chains are not bug-free [201]. For instance, *Xcode Ghost*[4] (a malicious version of Xcode), silently inserted malicious code at compile time and infected over 40 popular iOS applications to compromise millions of users' devices. Therefore, even if the source code is available, source code analysis sometimes falls short of revealing subtle bugs, security vulnerabilities, and malicious behavior. The reason is that subtle but important differences between programmer's intent and what

[2]https://inside.battelle.org/blog-details/hardware-vs.-software-vulnerabilities#:~:text=Hardware%20Vulnerabilities&text=Hardware%20vulnerabilities%20are%20more%20difficult,attackers%20slower%20to%20adopt%20them. Accessed on Jan 21, 2021.

[3]https://www.capita.com/sites/g/files/nginej146/files/2020-08/Ponemon-Global-Cost-of-Data-Breach-Study-2020.pdf. Accessed on Jan 21, 2021.

[4]https://us.norton.com/internetsecurity-emerging-threats-ios-malware-xcodeghost-infects-millions-of-apple-store-customers.html. Accessed on Dec 20, 2020.

is actually executed by the processor can be introduced during the code transformation and compilation process; this is called the WYSINWYX (What You See Is Not What You eXecute) phenomenon [20].

Detecting vulnerabilities through binary analysis in the presence of source code becomes less challenging and more accurate. In such cases, we can first conduct source code analysis, which is compiler and platform agnostic, to scan any code regardless of the underlying operating systems, development environments and compilers. This leads to a less challenging problem with more accurate results in the presence of various platforms. Moreover, source code analysis can be used to fix certain vulnerabilities even before the build stage. Also, locating vulnerabilities in the development stage has significant financial benefits and substantial savings in resources and time. In addition, we can compile the source code and keep the debug information (e.g., function identifiers) to preserve as much as information at the binary level. Therefore, in a perfect scenario the combination of both source code and binary code analysis would benefit the most to: (i) learn from two different views (e.g., source and binary) of the same vulnerability; (ii) detect subtle vulnerabilities or malicious code injected by attackers that will be present in the compiled version of a code; and (iii) reduce the false positive rates by including the standard libraries and free open-source libraries into the analysis during the compile time, and by comparing the obtained results from the analyses at both source and binary code levels. However, in this thesis, we focus on a harder threat model, which mainly covers closed-source software and malware samples (where source code is usually not available).

Binary analysis approaches are mainly categorized into three groups: static analysis, dynamic analysis and symbolic execution. More specifically, *static analysis* examines the assembly code or reconstructed source code of a given binary program without executing the code. *Dynamic analysis* is the process of examining the program behaviour while executing it in a controlled environment. In *symbolic execution* techniques, the program is

executed in an emulated environment using symbolic values rather than actual values in order to explore potential paths and find various kinds of errors, such as security vulnerabilities and memory corruption.

In recent years, binary analysis has been a promising solution for a wide-range of applications [4], such as threat analysis, vulnerability testing, digital forensics, malware analysis, recognizing copyright infringement and plagiarism detection. Furthermore, binary analysis is useful for security researchers and reverse engineers, since it offers important insights about a given binary code, such as information about its development environment and compilation, revealing its obscure functionalities, attribution of its author(s), recognizing standard and reused libraries, and identifying bugs and vulnerabilities. A typical binary *static analysis* approach starts with the process of disassembling (which is the process of obtaining the assembly code from the binary code), as a primary step towards understanding the behavior of a software and further identifying the presence of bugs and vulnerabilities. However, this process is tedious and time-consuming, and its success depends heavily on the experience and knowledge of binary analysts (a.k.a. reverse engineers). In the following, we briefly discuss the challenges, which are later elaborated upon in Chapter 2.

- Binary code analysis is more challenging when the source code is unavailable. Since a substantial amount of important information (e.g., variable and function names, types, data structures and control constructs) might be lost (or at least transformed) during the compilation process.

- Binary code analysis becomes significantly difficult due to the effects of code transformation. Code transformation is performed during compilation process, in which different compiler families and optimization settings, hardware architectures (e.g., x86 and ARM), and underlying operating systems contribute to this process. This becomes more challenging if further obfuscation techniques [220] are utilized.

- Today's binary code analysis heavily relies on manual efforts with a limited support

from automated tools, such as IDA PRO[5] [73] and GHIDRA[6]. Such manual analysis is typically tedious and error-prone, since a binary code inherently lacks any specific structure mainly due to the heavy use of jumps and symbolic addresses, control flows are highly optimized, and there exist various registers and memory locations that are processor and compiler dependent [21]. Moreover, analyzing large-scale binary code units involves additional overhead.

In what follows, we detail several important gaps in the literature of binary analysis, such as compiler provenance attribution, library function identification, vulnerable function detection and code similarity detection.

- Compiler provenance information can aid binary code and tool chain analysis by uncovering fingerprints of development environment and compiler functions. This information will accelerate the analysis by shortlisting the non-compiler functions. The existing techniques derive the compiler signatures from the meta-data or other details of program headers. This can be problematic as such information might be unavailable in stripped binaries or might be easily altered.

- Library function identification not only enhances the efficiency of reverse engineering and threat analysis tasks, but also improves their accuracy by avoiding false correlations between irrelevant code bases. The existing approaches may fail to identify the correct library functions due to slight modifications in the source code or the library version, mainly due to their reliance on the signatures derived from a limited number of features.

- With the growing popularity of the Internet of Things (IoT), the necessity of vulnerability detection and open-source usage in the firmware of IoT devices is more than ever. However, there exist few efforts to this end, while those techniques have several

[5] https://www.hex-rays.com/products/ida/support/download_freeware. Accessed on Dec 20, 2020.
[6] https://github.com/NationalSecurityAgency/ghidra. Accessed on Dec 20, 2020.

limitations related to their efficiency and accuracy. Moreover, none of them focuses on the detection of vulnerabilities in IoT devices in critical infrastructures, such as smart grids. Additionally, there is no large-scale databases (e.g., firmware images and their vulnerabilities) available for such research.

- With the evolution of heterogeneous systems, such as IoT and clouds, supporting code similarity detection in binary code and firmware images for various hardware architectures and compilers has become an essential requirement. However, none of the existing efforts targets code similarity detection and vulnerable function detection in cross-architecture obfuscated binaries.

In summary, compiler attribution, library function detection, vulnerable function detection and code similarity detection for cross-architecture and cross-compiler binary code and firmware images are essential open problems in binary analysis.

1.2 Problem Statement

This section enumerates the research questions of this thesis and defines our threat model.

1.2.1 Research Questions

This thesis aims at addressing several important research questions in statically analyzing and fingerprinting binary codes towards an investigation on the function identification in program binaries and applying the study to a broader range of applications (e.g., from desktop software to IoT firmware). To this end, this thesis mainly addresses the following key research questions:

"*How can we build a robust, scalable, and accurate system that identifies and fingerprints different types of functions in real-world and large-scale binaries (including malware and IoT firmware) for different platforms, compilers and hardware architectures?*"

More specifically, the main problem that this thesis aims to solve can be stated with the following major research questions:

1) How can we attribute compiler provenance in binaries compiled with different compilers and compilation settings on the x86 architecture?

2) How can we identify library functions in binaries compiled with different compilers and compilation settings on the x86 architecture?

3) How can we devise a vulnerable and open-source library function detection approach for program binaries and firmware images of IoT devices on the ARM architecture?

4) How can we detect vulnerable and open-source library functions to support multiple CPU architectures (e.g., x86 and ARM) for cross-compiled and obfuscated binaries?

In summary, these four research questions of this thesis focus on accuracy, robustness, and scalability aspects of function detection in real-world large-scale binaries (including malware and IoT firmware) written in C/C++ for different platforms (operating systems, compilers and hardware architectures). These topics are complementary to each other, as detailed in the following.

Given a binary code and passing it through a disassmebler, the binary code will be decomposed into a list of functions. These functions are originated from various sources, including compilers, standard libraries, free open-source libraries and user-defined functions. Moreover, some of the functions might have security vulnerabilities possibly as a result of code reuse practice. This thesis contributes towards identifying different types of functions in order to fingerprint a given code and further help assess its security. We start by attributing compiler provenance and identifying compiler functions. Then, we build an approach to detect standard library functions. Afterwards, we perform code similarity detection (a.k.a. clone detection) by identifying reused free open-source library functions

and further mark out those that are matched with the vulnerable functions in our repository. Finally, we enhance our code similarity detection and vulnerable function detection capability by supporting additional hardware architectures and overcoming the effects of obfuscation techniques.

Consequently, attributing compiler provenance, labelling compiler-related functions, and detecting library functions in a given unknown binary code will: (i) aid in code fingerprinting and characterization through providing information about the compilation process, underlying functionalities (based on the types of identified open-source libraries), etc. (ii) help the analysts shift focus to unknown and user-defined functions, and (iii) further contribute to the accuracy and efficiency of binary code analysis task, which is resulted from revealing the aforementioned information. Moreover, in order to examine the software security of a given binary code or embedded device firmware image, we identify the vulnerable functions that are high-likely borrowed from free open-source libraries.

1.2.2 Threat Model

In the following, we define our high-level threat model. This work does not require access to the source code of targeted system; instead, it relies on the availability of binary code. Consequently, we assume that the integrity of original binaries is preserved and the utilized disassemblers (e.g., IDA PRO) accurately perform function boundary identification and control flow graph generation. We also assume that binaries may contain known vulnerabilities reported by publicly available sources (e.g., CVE database), which usually introduced by programmers during the software development as a results of code reuse practice (e.g., free open-source libraries). Moreover, in this work we focus on anti-disassemblers obfuscation techniques [219], which includes a variety of techniques, such as dead-code insertion and control flow obfuscation. More specifically, we mainly consider the OBFUSCATOR-LLVM [120], which performs *instruction SUBstitution* (SUB), *Bogus*

Control Flow (BCF), and *control flow FLAttening* (FLA) obfuscation techniques. The reason behind this choise is its frequent use in the state-of-the-art approaches that eases the comparison.

This thesis (as part of feature identification and methodology design) considers several strategies that may affect detection mechanisms as follows. First, programmers or malware writers may copy an open-source library code and slightly modify it (while preserving its functionality) before code reuse practice. Second, programmers or malware writers may use different compilers or compilation settings, which affect the code representation of a program binary. Finally, anti-debugging obfuscation techniques (e.g., bogus control flow) might be utilized by programmers to protect the intellectual properties from being reverse engineered and released with malicious/vulnerable code, or by malware writers to hinder malware detection process. Thus, our solution is not specifically designed to overcome the hurdles imposed by binary packing, heavy obfuscation, or encryption. Instead, our system focuses more on identifying functions in binaries that are already unpacked, de-obfuscated, and decrypted using existing tools [74, 150]. Therefore, our proposed system is to assist the tasks of threat analysts and reverse engineers and not to replace them.

1.3 Research Contributions

To solve the stated problems, we follow four threads of research as detailed below. These proposed solutions are mainly based on the static analysis techniques.

1.3.1 Survey on Static Binary Analysis Approaches

We provide a comprehensive review of the state-of-the-art binary analysis solutions, which perform function matching and/or detect vulnerabilities in normal binaries and firmware images by employing static analysis, symbolic execution, and code similarity detection.

Furthermore, we perform both quantitative and qualitative comparisons amongst the surveyed approaches. Moreover, we devise taxonomies based on the applications of those approaches, the features used in the literature, and the type of analysis. Finally, we identify the unresolved challenges in this field of research. This survey has been accepted to be published in ACM Computing Surveys (CSUR). The details of this contribution is presented in Chapter 2.

1.3.2 Compiler Provenance Attribution

Compiler provenance encompasses several pieces of information, such as compiler family, compiler version, optimization level, and compiler-related functions. In order to attribute compiler provenance in binaries that are compiled with various compilers (e.g., GNU Compiler Collection (*GCC*) and Microsoft C++ (*MSVC*)) and their different compilation settings, we devise a practical approach called BINCOMP. BINCOMP is a multi-layered approach, which analyzes the syntax, structure, and semantics of disassembled functions. More specifically, it first applies a supervised compilation process to a set of known programs to model the default code transformation of compilers. Second, it employs an intersection process on disassembled functions across various sets of compiled binaries to find common compiler/linker-inserted functions and extract compiler-related features. Finally, it extracts semantic features from the labelled compiler-related functions to identify the compiler version and the optimization level. We evaluate the proposed approach on a large set of real-world binaries across several compiler families, versions, and optimization levels. The obtained results demonstrate that compiler provenance can be determined with high accuracy. The results of this work has been published in Digital Investigation [180]. The details of this approach are explained in Chapter 3.

1.3.3 Library Function Identification

In order to identify standard library functions in binaries that are compiled with different compilers and optimization settings, we develop a new technique called BINSHAPE. Our key idea is to derive the *shape* of each function, which is a novel concept based on a set of heterogeneous features. More specifically, BINSHAPE first derives the function shapes based on heterogeneous features, such as graph features, instruction-level features, statistical features, and function-call graph features. Second, it ranks those features using feature selection evaluators, such as mutual information-based ranking and decision trees, in order to obtain a signature for each library function. Finally, it extracts the signatures of known library functions and stores them in a repository using a novel data structure for efficiently matching against a target function. We evaluate BINSHAPE on a diverse set of binaries compiled with different compilers and optimization settings on x86-x64 CPU architecture. Our experiments demonstrate that BINSHAPE can identify library functions in real-world binaries both accurately, with an average accuracy of 89%, and efficiently, taking on average 0.14 s to identify one function out of three million candidates. Furthermore, BINSHAPE is robust enough when the code is subjected to different compilers or light obfuscation techniques. The results of this work has been published in the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA) [190]. The details of this approach are explained in Chapter 4.

1.3.4 Vulnerability Detection in Firmware Images

In order to detect vulnerable functions in program binaries and firmware images of IoT devices, we build BINARM. BINARM is a scalable approach to detecting vulnerable functions in binaries and smart grid intelligent electronic devices (IED) firmware, mainly based on the ARM architecture. To this end, BINARM first builds comprehensive databases of

vulnerabilities and firmware images that are widely used in IEDs in the smart grid. Second, it devises a multi-stage detection engine to minimize the computational cost of function matching and to address the scalability issue in handling vulnerability detection at large-scale. Specifically, the proposed engine takes a coarse-to-fine grained multi-stage function matching approach by (i) filtering dissimilar functions based on a group of heterogeneous features; (ii) dropping dissimilar functions based on their execution paths; and (iii) identifying candidate functions based on fuzzy graph matching. Our experimental results ascertain the performance of proposed system, which corresponds to an average total accuracy of 0.92. In addition, our study confirms the real-world applicability of BINARM, which successfully detects 93 potential common vulnerabilities and exposures (CVEs)[7] amongst real-world IED firmware within 0.09 seconds per function on average, the majority of which have been confirmed by our manual analysis. The results of this work has been published in the International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA) [189]. The details of this approach are explained in Chapter 5.

1.3.5 Cross-Architecture Code Similarity and Vulnerability Detection

To further expand our system to support cross-architecture binaries, we present TIOHTIÀ:KE that performs code similarity detection to identify vulnerable functions in a given binary code compiled with different compilers for different architectures. It leverages a neural machine translation approach that employs the Long Short-Term Memory (LSTM) Encoder-Decoder architecture [105], and models the assembly codes of a function as a sequence of instructions (similarly as the sentences in a natural language). It regards the assemble codes of two architectures (e.g., x86 and ARM) as two natural languages (e.g., English and French). Moreover, our utilized approach along with our features, makes TIOHTIÀ:KE

[7] https://cve.mitre.org/. Accessed on Dec 20, 2020.

more resilient to code obfuscations than state-of-the-art approach [69]. We perform extensive experiments on a large vulnerability dataset and obfuscated binaries to demonstrate the accuracy and efficiency of our approach. Furthermore, we apply TIOHTIÀ:KE to real-world firmware images. Performed experiments demonstrate that TIOHTIÀ:KE can successfully identify potential CVEs. The details of this approach are explained in Chapter 6.

1.3.6 Summary

In summary, the main contributions of this thesis are as follows.

- As per our knowledge, we are the first to propose a compiler provenance attribution approach that can simultaneously achieve several goals. This includes compiler family identification, compiler-related function labelling, compiler version detection, and optimization level recognition. Unlike existing solutions, the proposed approach derives the signatures only by relying on the characteristics of the binary, which are available even in the stripped form.

- We employ a diverse collection of features, including graph features, instruction-level characteristics, statistical characteristics, and function-call graphs, for library function identification. Consequently, our novel concept of the function *shape* induces a single robust signature based on heterogeneous features, which allows our proposed approach to obtain high accuracy. This is valid even when the code is compiled with different compilers and compilation settings, and is subjected to slight modifications due to obfuscation techniques.

- We propose a multi-stage detection engine to efficiently identify vulnerable functions from large-scale databases, while maintaining high accuracy. To this end, our proposed method is three orders of magnitude faster than the existing fuzzy matching

approach [110]. Furthermore, in this research, we develop the first large-scale vulnerability database (specifically for IEDs firmware covering most of major vendors). This vulnerability database can potentially be beneficial to IED vendors as well as utilities in order to assess the security of elaborated and deployed IED firmware.

- We propose a cross-architecture cross-compiler code similarity detection approach to identify vulnerable functions that can also handle obfuscation effects. First, we build our vulnerability database containing vulnerable and obfuscated functions cross-compiled for both ARM and x86 architectures. Furthermore, we introduce a new function representation including features that are less affected by code transformation techniques. Moreover, we adapt neural machine translation models to translate functions from one architecture (e.g., ARM) into another (e.g., x86). The proposed approach can identify known vulnerabilities from one architecture in binaries compiled for another architecture.

- We build a practical system with a user-friendly interface by integrating our proposed methods on vulnerability detection.

1.4 Thesis Organization

The remainder of this thesis is organized as follows.

- Chapter 2 provides a detailed review on related works in the area of static binary analysis, with the highlights of the current challenges, existing features and methodologies, and their taxonomies and comparisons.

- Chapter 3 details a practical approach, namely, BINCOMP, to attribute compiler provenance which can be used to analyze the syntax, structure, and semantics of disassembled functions in order to ascertain compiler provenance.

- Chapter 4 presents a scalable and robust library function identification approach, namely, BinShape, which utilizes a wide-range of features along with a novel data structure for scalable function indexing and searching.

- Chapter 5 describes a scalable and efficient vulnerable function detection technique, namely, BinARM, which can be used for vulnerability detection in real-world firmware images (e.g., IEDs) and binary programs.

- Chapter 6 presents a cross-architecture and cross-compiler framework for code similarity detection, namely, Tiohtià:ke, which provides an accurate solution to vulnerable code detection by employing deep neural machine translation techniques.

- Chapter 7 provides the concluding remarks along with a discussion on potential future work.

Chapter 2

Literature Review

This chapter first presents an overview of binary analysis and its challenges. Then, it discusses various features that are used in different binary analysis approaches and proposes a taxonomy of those features and corresponding existing approaches. It further presents potential application domains of binary analysis. Afterwards, it reviews existing static-based binary analysis approaches and devises a taxonomy based on three types of analysis employed in reviewed approaches, including graph-based, data flow-based and distance-based. Furthermore, it compares those approaches based on different criteria, e.g., methodologies, implementations, and evaluations.

2.1 Binary Analysis Overview and Challenges

In this section, we first introduce the major approaches (e.g., static analysis, dynamic analysis, and symbolic execution) in binary analysis and then describe different challenges of binary analysis.

2.1.1 Overview

Binary analysis can be performed by inspecting a code statically, or executing a program dynamically or with symbolic values. These approaches are called *static analysis*, *dynamic analysis* and *symbolic execution*, respectively, as discussed in the following.

Static Analysis. Static analysis examines a given binary code rather than executing it. It is typically designed to reason about the entire program, and has the capability to explore all potential execution paths of a given code. The *code similarity* problem focuses on analyzing two pieces of binary code (e.g., functions) in order to measure their similarities. A function is deemed as a potential match, if there is a match between the function and an already analyzed function (e.g., a vulnerable function) in the repository. Various static solutions based on code similarity detection on both program binaries and firmware images have been proposed in the literature.

Dynamic Analysis. Dynamic analysis is the process of examining and monitoring the program behaviour while it is running in a controlled environment. For instance, dynamic analysis can be performed by automatically generating malformed or user controlled inputs as a tainted data. Moreover, emulation-based techniques build partial/full simulation for a specific architecture/platform, and then employ powerful and advanced dynamic analysis techniques to run the software in the simulated environment.

Symbolic Execution. Symbolic execution techniques allow to reason about the behavior of a program on many different inputs at one time. Instead of reading concrete values (e.g., 7) during a normal execution, symbolic values (e.g., λ) are utilized as inputs. These techniques aim at reaching a specific program state through generating the inputs that satisfy the required path constraints. Therefore, symbolic execution can explore all potential paths compared to concrete execution, which can explore only one path that is related to the supplied concrete inputs at a time. However, symbolic execution techniques suffer from reliance on computationally expensive solvers (e.g., [26]) as well as path explosion.

2.1.2 Challenges

In the following, we outline the most critical challenges that complicate the binary analysis process in the absence of source code.

- **C1 -** *Information loss*: During the compilation process, some information that is available in the source code, ranging from syntax features (e.g., variable names and comments), to characteristics of the buffers and data structures sizes will be lost. Therefore, analyzing a binary code would become more challenging and complicated compared to source code analysis. Additionally, in the case of *stripped binaries* where the debugging information (e.g., identifier names) is missing, binary analysis task becomes more challenging.

- **C2 -** *Compiler effects*: With the advent of modern compilers and run-time libraries, binary code analysis is becoming a very challenging task. Most compilers apply performance or memory optimizations, which result in significant variations in binary representations. These variations may include different registers, calling conventions, control flow graphs, and mnemonics and arithmetic operations. These differences are more significant if another compiler or compilation settings are used.

- **C3 -** *Binary disassembling*: Binary disassembling is still a challenging task mainly due to the following reasons:

  – *Entry point and function boundary discovery*: The disassembler usually uses a *symbol table* to identify function boundaries and to construct the control flow graphs. However, when the *symbol table* is inaccurate or it is not available, finding function boundaries becomes challenging [221]. Additionally, in some cases such as *binary-blob* firmware [192], the entry point and the base address are not known [210]. Moreover, some functions may have multiple entry points [23], which need to be identified.

– *Code discovery*: To align instructions and improve cache efficiency, compilers may insert padding bytes between or within the functions. Consequently, the disassembler may not distinguish padded bytes from code bytes, since padded bytes are usually converted into valid instructions [23]. Moreover, some functions may not be continuous and could have some gaps including data, jump tables, or instructions from other functions [23], which affect the accuracy of the binary analysis approaches. Additionally, some binary analysis tools fail to accurately build control flow graph (CFG) due to their inability to find all the code, identifying non-return functions, and handling indirect jumps. Some of these issues might be resolved under the approximations and heuristics that could lead to false positives. As a result, any analyses which rely on the CFGs would be directly affected by imprecise CFG generation. However, some solutions are proposed to overcome these limitations, for instance, value set analysis (VSA)[1] [20] is employed to resolve the indirect jumps.

– *Code transformation*: The authors of legitimate/benign programs may protect their programs for different reasons, such as intellectual property copyright infringement or preventing their programs from being repackaged and redistributed as malware [2, 211]. Similarly, malware authors apply related techniques on their malicious code to evade analysis and make them more cumbersome. Obfuscation [52, 130, 220], encryption and packing [121, 199] techniques are used for these purposes to make the binary analysis more challenging and difficult.

- **C4 -** *Function inlining:* A small function might be inlined into its caller function for optimization purposes. The lack of distinction between an inline function and the

[1] VSA is a numeric and pointer-analysis algorithm that determines an over-approximation of the set of numeric values and addresses (or value-set) that each abstract location holds at each program point.

other parts of the function makes the function inline identification task very challenging. This task becomes even more challenging when the assembly instructions of an inline function are discontinuous as the result of instruction alignment and pipelining.

- **C5 -** *Hardware architecture*: Software programs can be cross-compiled or deployed on different CPU architectures, where instruction sets, calling conventions, register sets, function offsets and memory access strategies vary from one architecture to another [174]. Therefore, analyzing binaries compiled for different CPU architectures but originated from the same source code is more challenging.

- **C6 -** *Accuracy*: Achieving high accuracy for any function detection technique is a critical and non-trivial task. For instance, obtaining low false positive rates is usually challenging when analyzing the code statically [191].

- **C7 -** *Results verification*: For several techniques, the obtained vulnerability detection results cannot be verified due to the limited access to specific information on the identified vulnerabilities (e.g., how to trigger). Therefore, these techniques involve manual efforts to verify the results and hence, can be error-prone and inefficient.

- **C8 -** *Efficiency*: Many existing approaches are computationally expensive, which highlights the need of efficient identification techniques for any binary code.

- **C9 -** *Scalability*: Due to the dramatic growth of desktop applications, IoT devices, and inter-connectivity between them, the number of deployed software is increasing exponentially. As such, function detection approaches need to deal with a large number of binaries and firmware images, which indicates that large scale binary analysis is an absolute requirement.

- **C10 -** *Test case generation*: Some approaches require an initial input seed compatible with the target application to properly start with the analysis. Test cases are

easy to generate when the targeted application provides its required input file format. However, when no configuration input is provided, test case generation becomes a challenging task; since each program needs a particular test case to be prepared in advance, which requires expertise and additional effort.

- **C11-** *Firmware reverse engineering:* Firmware reverse engineering can be a time consuming and challenging task that requires domain expertise. The whole process involves the following steps:
  - *Firmware acquisition*: Embedded system vendors tend to avoid publishing their firmware in order to limit accessing it and to protect their intellectual property. Therefore, it might be necessary to directly extract or dump it from a device chip memory in different ways, such as an EEPROM programmer, bus monitoring during code upload and schematic extraction [203]. However, hardware locks and component interference might make this task challenging. This can be resolved by physically modifying the original hardware, or manipulating the circuit boards using probes.
  - *Firmware unpacking and extraction*: Some vendors pack their firmware using proprietary packers and file formats, or use private key encryption. In practice, different unpacking tools, such as BINWALK[2], BAT [103], and FRAK [57] can be utilized to extract the firmware. However, performing such tasks has limited success rate and thus not all embedded device firmware can be analyzed (e.g., Costen et al. [44] successfully unpacked $8,617$ firmware out of $23,035$ collected firmware images).
  - *Firmware and binary identification*: Once the firmware images is unpacked, filtering is required for obtaining all relevant information. This can include binary files, configuration files, embedded files and the firmware itself. To this end, file

[2] https://github.com/devttys0/binwalk. Accessed on Dec 20, 2020.

signature matching is performed using different tools, SIGNSRCH[3], FILE[4], and BINWALK. Also, there exist some types of firmware that have no underlying operating system. They consist of only one binary file that operates directly on the hardware. In some cases, there is no abstraction of the OS and libraries, and in other cases, firmware images are not standard and no documentation is provided. Therefore, initializing a run-time environment and loading the binary is more challenging [192].

Summary. Static analysis techniques usually suffer from the C1-C9 and C12 limitations. For instance, such techniques may misidentify non-vulnerabilities, leading to high false positive rates [191], or they may fail to find all the vulnerabilities (e.g., run-time vulnerabilities), generating more false negatives. Additionally, since the information to trigger the identified vulnerability is not provided, the results of vulnerability detection should be verified manually. Although dynamic analysis techniques can overcome some challenges (e.g., C2, C3 and C7) of static analysis approaches, they are still affected by the C5, and C8-C11 limitations. On the other hand, static analysis approaches have their own advantages, e.g., they are scalable compared to dynamic analysis approaches. Therefore, researchers recently combine static analysis with dynamic analysis approaches. The symbolic execution techniques suffer from the C8 and C9 limitations.

2.2 Preliminaries for Static Approaches

This section provides a discussion on several preliminaries in static approaches, which is the primary focus of this thesis. These include intermediate representation, feature extraction, and application domains.

[3] http://aluigi.altervista.org/mytoolz.htm. Accessed on Dec 20, 2020.
[4] https://linux.die.net/man/1/file. Accessed on Dec 20, 2020.

2.2.1 Intermediate Representation

Intermediate representation (IR) is a processor-neutral form, which represents the operational semantics of a binary code with an intermediary level of abstraction. In order to support multiple CPU architectures, the disassembled binary codes are converted to an intermediate representation; this process is also called binary lifting. Designing an intermediate lifter is a difficult task, since the instruction set manual of each CPU architecture comprises thousands of pages. Additionally, the CPU instruction semantics are usually not sufficiently detailed and some of them are undefined or undocumented, which leads to a trial-and-error based process to lift the instruction sets to IR representation. Therefore, binary analysts and reverse engineers generally reuse available binary lifters. However, the accuracy of intermediate lifters directly affects the accuracy of the underlying binary analysis techniques. Consequently, different factors are taken into account when choosing the appropriate intermediate lifters. (i) *Is the lifter open-source?* (ii) *How accurate is the lifter?* (iii) *How many CPU architectures (e.g., x86 and ARM) are supported?* (iv) *Are there any dependencies for any specific disassembler?* (v) *Does the lifter support floating point and Single Instruction Multiple Data (SIMD) instructions?*

Kim et *al.* [126] study the characteristics of existing open-source intermediate representations, such as LLVM [135] (called REMILL-IR[5]) and QEMU[6] [27] (called TCG-IR[7]). They further evaluate the efficiency of three well-know open-source lifters: BAP [33] (called BIL-IR[8]), BINSEC [24] (called DBA-IR[9]), and PYVEX [192] (called VEX-IR[10] [166]). For this purpose, the authors design and implement a tool called, MEANDIFF[11], which supports x86 and x86-64 architectures solely in order to find the semantic bugs in

[5] `https://github.com/trailofbits/remill`. Accessed on Dec 20, 2020.
[6] `https://git.qemu.org/?p=qemu.git`. Accessed on Dec 20, 2020.
[7] `https://github.com/qemu/qemu/tree/master/tcg`. Accessed on Dec 20, 2020.
[8] `https://github.com/BinaryAnalysisPlatform/bil`. Accessed on Dec 20, 2020.
[9] `https://github.com/binsec/binsec/tree/master/src/dba`. Accessed on Dec 20, 2020.
[10] `https://github.com/angr/pyvex`. Accessed on Dec 20, 2020.
[11] `https://github.com/SoftSec-KAIST/MeanDiff`. Accessed on Dec 20, 2020.

binary lifters. They observe that none of the lifters is completely precise, since all of them fail to lift hundreds of instructions. According to their evaluation on the three state-of-the-art lifters, BAP and BINSEC lift successfully a large number of instructions compared to PYVEX. They observe that BAP and BINSEC are able to handle loop statements-related instructions (e.g., `f3:rep` prefix instruction) while PYVEX cannot handle them. However, BAP and BINSEC do not have full support of floating-point and SIMD instructions. Furthermore, BINSEC does not support x86-64 CPU architecture. On the other hand, PYVEX supports five different CPU architectures, as well as floating-point and SIMD instructions. Consequently, the obtained results indicate that the accuracy of any IR-based approach is directly related to the accuracy of the underlying lifter [126].

2.2.2 Feature Extraction

In order to analyze a given binary code statically, various features from different binary code representations or intermediate representations (IR) can be extracted. We classify the features into four categories of *instruction-level*, *statistical*, *structural*, and *semantic* features. The proposed feature taxonomy as well as the corresponding approaches (explained later) that utilize these features are illustrated in Figure 2.1.

Instruction-level Features. Instruction-level features can be extracted directly from a given binary code. For instance, n-*grams* [163] are n sequences of tokens (e.g., bytes or instructions) in a program binary. One drawback of n-*grams* is that they are sensitive to the order of instructions, since some instruction could be reordered while semantics remains the same. To solve this issue, n-*perms* [122] are proposed, which are n sequences of tokens with any order. *Idioms* [186] are composed of short sequences of instructions with wildcards, where the values of immediate operands are abstracted away . A *mnemonic* [125] of an instruction represents the operations that need to be executed, whilst typically an *opcode*

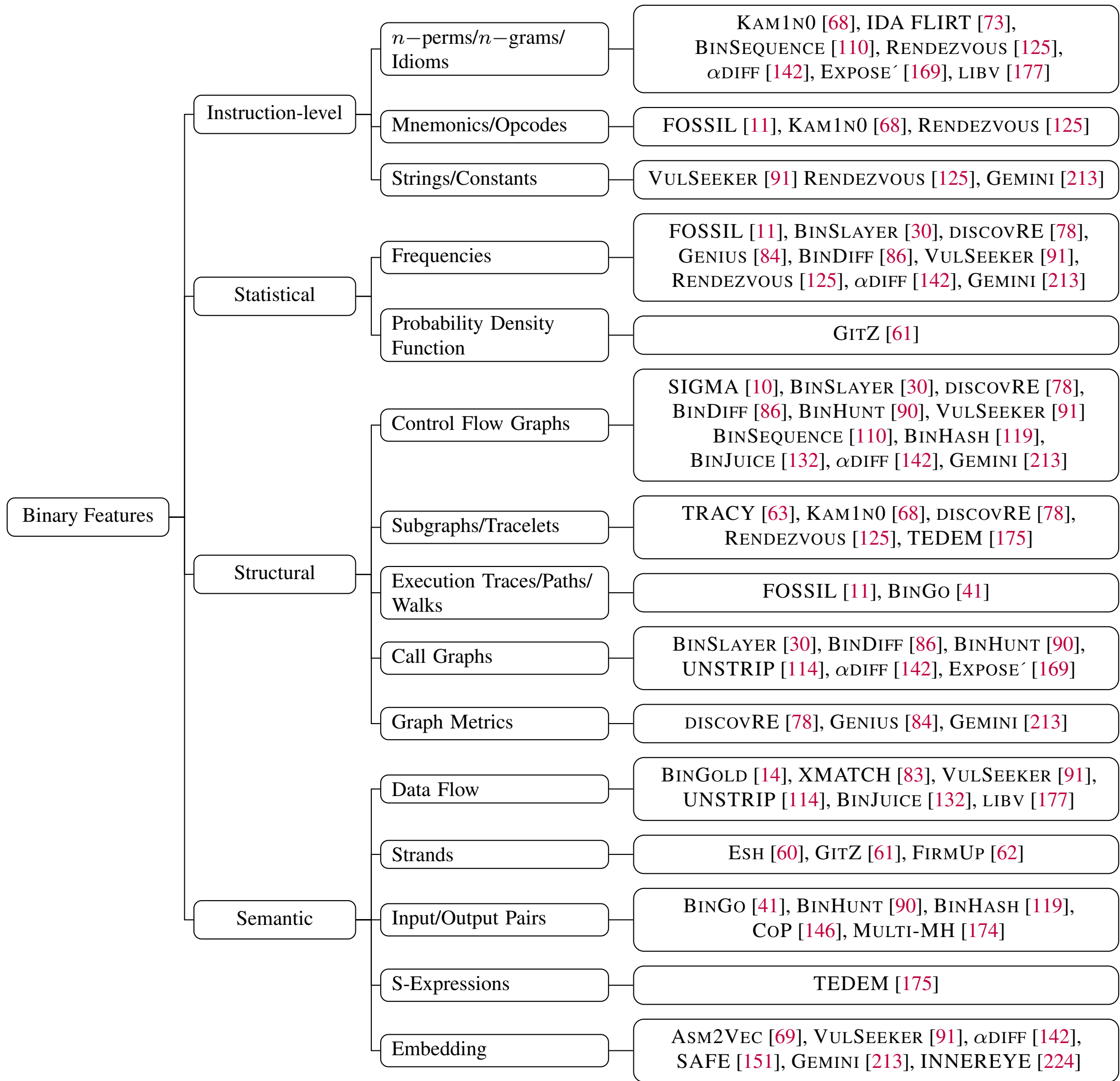


Figure 2.1: The proposed taxonomy of the features and corresponding existing approaches

is the hexadecimal encoding of the instruction . *Strings* and *constants* are other instruction-level features that can be captured easily from the instruction sets. The motivation of using constants as a feature is that usually constants remain unchanged regardless of compilers and optimization settings.

Statistical Features. Statistical features represent the semantic information of a binary code, for instance, cryptography functions use more arithmetic and logical instructions compared to a function that writes some information into a file. For this purpose, the

statistics of different features, such as *frequencies*, are used in the literature. As an example, instructions grouping [129] (e.g., number of arithmetic instruction) is utilized to get more information about the functionality of a function. *Opcode distributions* [28] are used to detect metamorphic malware. Additionally, some statistical distributions are computed to get more information about a function and the distribution of its instructions [61].

Structural Features. Structural features represent the semantic information as well as the structural properties of a function. *Control flow graphs* (CFGs) [86] are the most frequent features that are used in the literature. A subset of CFGs called *tracelets* [63], *partial traces* [41] or *subgraphs* [129], which are defined as the short and partial traces of an execution, capture the semantic of execution sequences. Similarly, *execution traces/paths/walks* [11, 41, 189] represent the execution traces, while considering basic block semantics. *Call graphs* [86] are extracted at the program level to get more information about both the relation between callers and callees as well as the program logic. Additionally, some graph metrics [96], such as *graph energy* and *betweenness centrality* (node centrality) [167], are used to extract more information about the topology of the graph.

Semantic Features. This category includes features that put more emphasis on conveying code semantics. The *execution flow graph* (EFG) [177] preserves the data and control dependencies across the instructions in a CFG and therefore represents the internal structure of a function. *Data dependence* and *program dependence graphs* (PDGs) [161, 191] allow to reason about the control and data flow, where memory and register values are required to be extracted. Data flow analysis combined with path slicing [117] and value-set analysis [18, 20] are employed to construct *conditional formulas* [83], which describe when a given action will take place under which conditions and could capture incorrect data dependencies and condition checks. *Strands* [60] are the set of instructions resulting from backward slicing [209] at basic block level. *Input/Output pairs* [41, 90, 119, 145, 146, 174] are obtained by the assignment formulas for each basic block, where the effects of input variables

on the output variables are monitored in order to capture their semantics. *S-Expressions* [175] are a tree-like data structure composed of equations, which capture the effects of basic blocks on the program state. The equations are obtained from the basic block instructions, where both left-hand and right-hand sides contain arbitrary computations. *Side effects* [75] are composed of a set of features, such as values written to (or read from) program heap, system calls, etc., that are collected during the program execution and capture function semantics. *Embeddings* [213, 224] are high-dimension numerical vectors obtained from a function or a fragment of it (e.g., CFG or basic blocks), which preserve and convey the semantics of functions.

2.2.3 Application Domains

Binary analysis and code fingerprinting have been performed on different application domains, including (i) *authorship attribution*, which refers to the detection and attribution of the author(s) of a given binary code; (ii) *compiler provenance attribution*, which provides information about the utilized compiler family, version, optimization level and compiler functions; (iii) *library function identification*, which identifies the standard library (libc) functions in a binary code; (iv) *code reuse detection (a.k.a clone detection)* that aims at recognizing similar fragments of two given code samples, which could be performed at program level or function level, such as free open-source software (FOSS) packages that are borrowed from available open-source libraries (e.g., `OpenSSL`); and (v) *vulnerable function identification* which is utilized to identify vulnerabilities in a given code. We categorize the existing works based on the aforementioned domains as presented in Figure 2.2.

Authorship Attribution. The task of authorship attribution aims at identifying and characterizing the author or a group of authors for a given binary [3]. In contrast to source code, binary code authorship attribution has drawn significantly less attention. This is mainly due to the fact that many salient author-related features that may identify an author's style

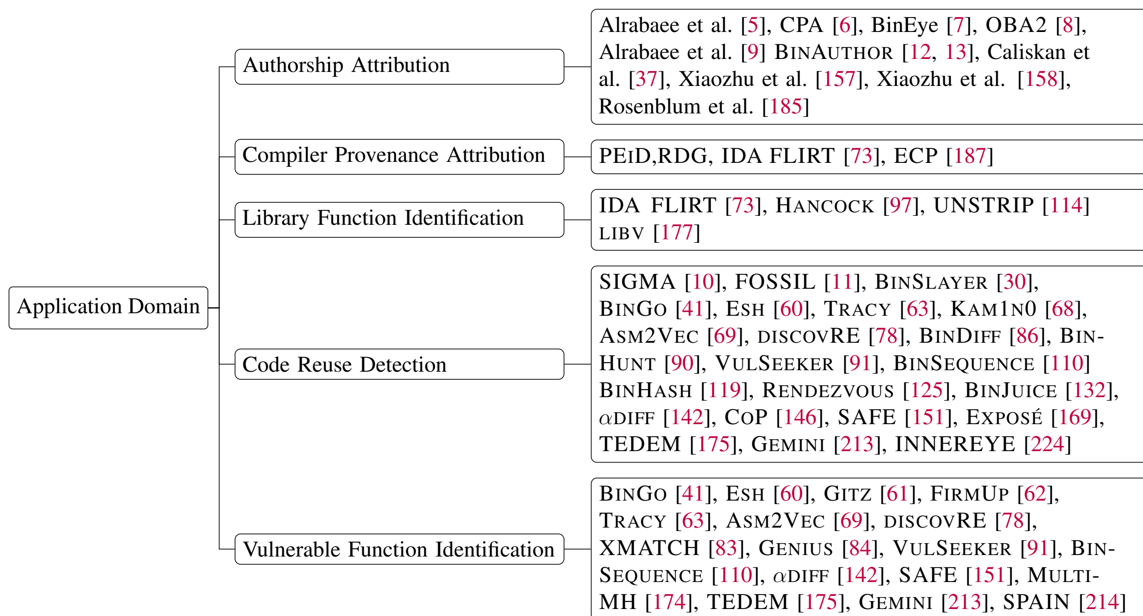


Figure 2.2: Taxonomy of application domains

are lost during the compilation process. In [8, 12, 13, 37, 185], the authors show that certain stylistic features can indeed survive the compilation process and remain intact in binary code, thus showing that authorship attribution for binary code should be feasible. In [157, 158], the authors introduce new fine-grained techniques to address the problem of identifying multiple authors of a binary code by determining the author of each basic block. Recently, convolutional neural networks (CNN) [94] are utilized to characterize the authors of program binaries [5, 7]. Furthermore, there are certain works that mainly perform manual analysis for malware authorship attribution, such as the technical reports published by *Citizen Lab*[12], *BlackHat* [149], and *FireEye* [160]. Since authorship attribution is not in the scope of this thesis, we refer the reader to [3] for more details.

Compiler Provenance Attribution. Few studies have been conducted on extracting compiler provenance [113, 184, 187, 202]. The existing body of work can be considered as a series of pioneering efforts, beginning with labelling functions in stripped binaries [113], followed by identifying the source compiler of program binaries [184], and culminating

[12] https://citizenlab.org/. Accessed on Dec 20, 2020.

in recovering the toolchain provenance of binary code [187]. Finally, in [202], a hidden Markov model is used to identify the compiler. Furthermore, there are certain tools that are capable of identifying compilers (e.g., PEiD[13], IDA Pro[14] and RDG[15]. Inspired by such efforts, we provide improvements in a prototype system called BinComp [180], as explained in details in Chapter 3.

Library Function Identification. Modern software typically contain a significant number of library functions, and identifying such functions in a binary file can offer a vital help to threat analysts and reverse engineers in many practical security applications. A *library function* is a function with known semantics and instructions. Therefore, by identifying such functions, there would be no need to reverse engineer them and the analysis can shift focus to unknown functions. The objective of this task it to identify library functions which are statically linked in a stripped binary.

Various approaches for library function identification have been proposed. The well-known library identification technique, IDA FLIRT (Fast Library Identification and Recognition Technology) [73], builds the signatures from the first 32 bytes of a function with wildcards for bytes that vary when the library is loaded. IDA FLIRT uses pattern-matching algorithms to determine whether a disassembled function matches one of the known signatures. FLIRT first tries to identify the compiler of a disassembled program, and then applies only signatures for that compiler. Therefore, this approach could lead to false negatives because of failure in detecting the compiler. In addition, FLIRT suffers from two main limitations: signature collision, and sensitivity to any slight differences in the code due to various factors, such as small variation in libraries (e.g., minor changes in the source code), different compiler optimization settings, or use of various compiler versions.

In contrast to FLIRT's conservative approach, Hancock's [97] primary goal is to

[13] https://github.com/wolfram77web/app-peid. Accessed on Dec 20, 2020.
[14] https://www.hex-rays.com/products/ida/. Accessed on Dec 20, 2020.
[15] http://www.rdgsoft.net/. Accessed on Dec 20, 2020.

eliminate false positive signatures. The main goal of HANCOCK is malware detection, however, HANCOCK employs a heuristic approach component to label the library functions. More specifically, HANCOCK improves the FLIRT technology by using three heuristics: (i) *Universal FLIRT Heuristic*, which matches a target function against all FLIRT signatures, regardless of the compiler recognition; (ii) *Library Function Reference Heuristics*, which labels a function that is statically called by any known library function as a library function; and (iii) *Address Space Heuristic*, which checks if the size of the space between a known library function and another function is below 128 bytes in order to identify the function as a library function.

Another approach, called UNSTRIP[16] [114], identifies the library functions in the GNU C library based on semantic descriptions, which are obtained from the interaction of wrapper functions with the system call interface. In order to capture the high-level semantics of wrapper functions and create the function fingerprint, a semantic descriptor based on the name and parameter values of the invoked system call is constructed. Backward slicing [50, 127] and symbolic evaluation [42, 56] are employed to extract system call names and the corresponding arguments in order to create a semantic descriptor for each wrapper function as a signature. Then, a flexible pattern matching is applied on the library fingerprints to identify the wrapper functions. However, UNSTRIP focuses on wrapper functions in Linux platform, and a library function may have no system call.

An approach to identify library functions and inline functions called *libv* is proposed in [177]. The authors first introduce *execution dependence graphs (EDGs)* to describe the behavioral characteristics of binary code. Then, by applying a graph isomorphism and finding similar EDG subgraphs in target functions, they identify both full and inline library functions. To improve the subgraph isomorphism testing, the authors reduce the execution flow graph (REFG). Additionally, to improve the efficiency, five filters based on the number

[16] `http://www.paradyn.org/html/tools/unstrip.html`. Accessed on Dec 20, 2020.

of instructions in a function, basic blocks lengths, number of basic blocks, basic block sequences and head-tail nodes are proposed. However, this approach has some limitations: (i) if various library functions have the same EDGs, this approach cannot distinguish them; and (ii) the compiler optimization settings might eliminate instructions of inline functions, which may lead to misidentification.

In this thesis, we propose BINSHAPE [190] in order to identify standard library functions as presented in details in Chapter 3.

Code Reuse Detection. In many software development environments, it is a common practice to borrow existing open-source code and libraries (e.g., OpenSSL), as this significantly reduces the programming effort and improves the efficiency. The first requirement for any binary code reuse detection system is the presence of a ground truth repository, where the fingerprints/signatures of the known functions are stored. Given an unknown binary code and a repository of already analyzed and labelled code, the objective of code reuse detection is to identify identical or semantically similar code with the code in the repository. Code reuse detection can be typically achieved by calculating the similarity score between two given pieces of code. The higher the similarity, the more likely they carry the same semantics. There exist different approaches in the literature for binary code similarity and clone detection, which are explained in details in Section 2.3.

Vulnerable Function Detection. A given binary code might contain vulnerable functions. One way to identify known vulnerabilities and CVEs[17] is to employ code reuse detection. In the code reuse process, if the borrowed code contains any bugs or vulnerabilities, the developers may bring the vulnerability into their own project. Library reuse is a special case in which the developers either include the source code of a certain library into their projects, or statically link to the library. Either way, the bug contained in the reused code will be brought into the new project. Thus, code reuse detection can help identify such

[17] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160. Accessed on Dec 20, 2020.

vulnerabilities resulting from source code sharing.

In this thesis, we propose BINARM [189] to detect open-source library functions and vulnerable functions in binaries and firmware images compiled for ARM architecture, as presented in Chapter 4. Moreover, we propose TIOHTIÀ:KE to identify open-source library functions and vulnerable functions in cross-architecture obfuscated binaries and firmware images, as detailed in Chapter 6.

2.3 Static Binary Analysis Approaches

In this section, we further describe and categorize existing static approaches (as this thesis primarily focuses on static solutions). Specifically, we classify the static approaches into three categories of: *graph-based*, *data flow-based*, and *distance-based*, as illustrated in Figure 2.3. Moreover, we elaborate on the corresponding existing state-of-the-art approaches, which are mostly proposed for function detection and more specifically vulnerability detection. Furthermore, we review symbolic execution approaches which are combined with static analysis. Finally, we compare static solutions both qualitatively and quantitatively based on various aspects, such as the features, methodologies, architectures, implementations and evaluations.

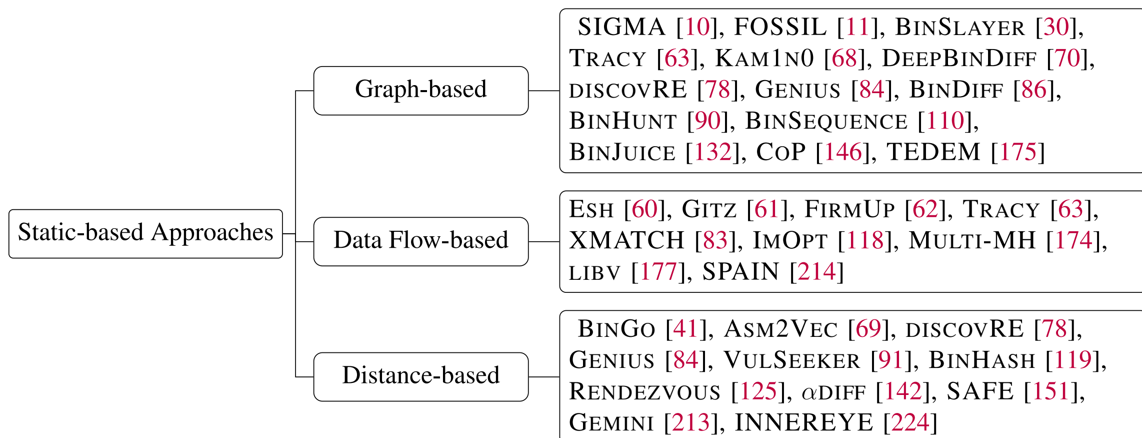


Figure 2.3: Taxonomy of static-based binary analysis approaches

2.3.1 Graph-Based Approaches

Graph-based approaches perform the analysis based on graph representation of a piece of code, such as control flowgraph (CFG), subgraphs, tracelets and call graphs, each of which convey specific information. The pioneering work BINDIFF[18] [72, 86] compares two different versions of the same binary. The executables are converted to a directed graph, where the functions and corresponding function calls are represented by the nodes and edges, respectively. Graph isomorphism is employed to match the similarity of the graphs. However, BINDIFF is not designed to identify vulnerable functions at large scale. Inspired by BINDIFF, BINSLAYER [30] utilizes a polynomial algorithm to compare two executable files, which is obtained by fusing the BINDIFF [72, 86] and Hungarian algorithm [162]. BINSLAYER finds matches with the minimum possible graph edit distance (GED) [92] over all functions and basic blocks, which provides a more robust matching algorithm.

Recently, an unsupervised learning approach called DEEPBINDIFF [70] is proposed to perform binary diffing. First, the `word2vec` [159] is applied on the instructions. Then, program-wide structural information are extracted from the inter-procedural CFGs (ICFGs) by employing the *Text-Associated DeepWalk* (TADW) algorithm [215]; this results in basic block level embeddings. Finally, the authors propose a k-hop greedy matching algorithm to obtain the optimal matching results for a given basic block.

A program (or code fragment) similarity detection approach called BINJUICE [132] extracts the effects of basic blocks on program state; these effects are termed as '*juice*'. The *juice* is obtained by symbolic interpretation of individual blocks, where the register names and constants are replaced by typed logical variables. Thus, the *juice* forms a semantic template that is expected to be identical regardless of code variations due to register renaming, memory address allocation, and constant replacement. Simple structural comparison or

[18] `https://www.zynamics.com/bindiff.html`. Accessed on Dec 20, 2020.

hash comparison of the *juice*s are employed to identify equivalent code fragments. However, if the semantics of two semantically equivalent programs are spread differently in their corresponding basic blocks, the extracted *juices* would not be similar. Thus, code transformation techniques (e.g, compiler effects) would affect BINJUICE's accuracy [132].

In order to measure function similarities, a framework called TRACY[19] [63] is proposed, which extracts partial execution traces (*tracelets*) from binary functions. TRACY breaks CFGs into tracelets and utilizes the longest common subsequence (LCS) algorithm [53, 207] in order to align the tracelets. It employs a rewriting engine rule to handle the alignment and data dependencies for comparing memory locations and registers between the tracelets using data flow analysis. Furthermore, TRACY models the rewrite problem as a constraint-solving problem to reduce the search space of possible rewrite sequence. Consequently, tracelets similarity are measured by counting the number of rewrite rules required to reach from one tracelet to another one. However, many structural information is lost by decomposing the CFGs into tracelets. Additionally, having different optimization settings affects their accuracy. Furthermore, the tool provides better results for large functions (e.g., with more than 100 basic blocks) [63].

Identifying the bugs based on the signatures of known security bugs is proposed in TEDEM [175]. First, the semantic signatures of each basic block are captured by the expression tree (*S-Expression*). Then, the tree edit distance (TED) [200] on the basic block combined with the Hungarian algorithm to explore the neighbours in a CFG are employed to detect a vulnerable function. A pre-filtering process based on the coarse-grained basic block attributes, such as number of equations, number of nodes, and depth of the tree is performed to tackle the scalability issue. However, tree edit distance slows down the matching process significantly, and not all the syntactical changes are captured [174].

Another approach called SIGMA [10] identifies reused functions in binary code by the

[19] https://github.com/Yanivmd/TRACY. Accessed on Dec 20, 2020.

graph-based representation of a code. Graph edit distance is employed for matching traces of the proposed semantic integrated graph (SIG), which is composed of the combination of CFG, register flow graph (RFG) [8], and function call graph. However, the time complexity of the detection algorithm is high, and the proposed approach is not designed for vulnerability detection in large scale dataset.

A search engine called KAM1N0[20] [68] is proposed to identify subgraph clones from a large code repository. Adaptive locality sensitivity hashing (LSH) [15] is used to find the basic block pairs, and then *MapReduce* [139] (based on the Apache Spark) is proposed to construct the subgraph clones by merging the clone block pairs. However, KAM1N0 is sensitive to instruction set changes and optimization settings. Additionally, it requires the assembly code of the same chosen family to be present in the repository [68].

A code reuse detection approach called BINSEQUENCE [110] detects code reuse by fuzzy graph matching along with employing the longest common subsequence (LCS) algorithm combined with Hungarian algorithm. The candidate functions are filtered using two filters: (i) number of basic blocks, and (ii) function fingerprint similarities based on the Minhashing [139] and banding techniques. However, code transformation techniques, such as compiler optimization, will affect its accuracy [110]. In addition, the proposed filtering affects the efficiency in the case of large and complex functions.

A resilient and efficient system to determine free open-source software packages is proposed in FOSSIL [11]. FOSSIL proposes three components and then integrates them using a Bayesian network model [176] in order to synthesize the results. The three components apply: (i) a hidden Markov model statistical test to opcode frequencies as syntactical features; (ii) a neighbourhood hash graph kernel [123, 206] to random walks to extract function semantics; and (iii) z-score to the normalized instructions to extract the behavior of the instructions in a function. The novel approach of combining these components using

[20] https://github.com/McGill-DMaS/Kam1n0-Community. Accessed on Dec 20, 2020.

the Bayesian network has produced strong resilience to code obfuscation.

To perform vulnerable function detection in cross-compiled cross-architectures binary code DISCOVRE [78] is proposed. It searches for known vulnerable functions amongst binary files cross-compiled for different CPU architectures and various compilers and optimization settings. It extends the maximum common subgraph (MCS) [154] distance to additionally consider the similarity between basic blocks based on different features, such as topological order in the function, strings, and constants. Since MCS execution time grows exponentially, the authors terminate the algorithm after a certain number of iterations. Furthermore, to reduce the cost of subgraph matching, a numerical filter based on a set of features (e.g., number of instructions, number of parameters, local variable sizes, and number of incoming/outgoing edges to/from a function) and the KNN algorithm [58] are employed. Nevertheless, according to the performed evaluation in [84], the utilized pre-filtering causes notable diminution in accuracy. DISCOVRE is tested on the firmware images of the DD-WRT router, NetGear ReadyNAS, and Android ROM image.

Inspired by DISCOVRE, a bug search engine called GENIUS [84] is proposed to identify vulnerable functions. GENIUS utilizes both statistical (e.g., number of calls and number of arithmetic instructions) and structural features (e.g., betweeness and number of offspring) that are consistent among multiple CPU architectures and then labels each basic block with a set of attributes to construct the attributed control flow graph (ACFG). To perform an efficient searching process, the ACFGs are converted into codebooks using spectral clustering [168] and further encoded [16] using a high-level embedding and locality sensitive hashing (LSH). The performed evaluation demonstrates that GENIUS can identify similar functions within one second on average in a large dataset. However, the authors state that creating the codebook is expensive which later also it has been demonstrated in [213]. The bug search is performed on $8,126$ firmware images from 26 different vendors, such as ATT, Verizon, Linksys, D-Link, Seiki, Polycom, and TRENDnet. This includes different products, such

as IP cameras, routers, and access points. Moreover, similar to DISCOVRE, GENIUS is evaluated on DD-WRT router, and NetGear ReadyNAS firmware images.

2.3.2 Data-Flow-Based Approaches

Data flow-based approaches typically observe the flow of data by analyzing the memory reads and writes, input/output pairs, variable locations, etc. However, most of the existing data flow-based solutions cannot be applied at large scale.

An approach called BINHASH [119] captures function semantics by representing the input/output behaviour of the basic blocks as a set of features using MinHashig. Further, the functions are clustered according to their hash values, where the functions in the same cluster are deemed to be similar. However, this approach is not scalable for basic blocks with large input/outout dimensions [175].

In order to identify vulnerable functions compiled over multiple CPU architectures, MULTI-MH [174] derives bug signatures in the form of subgraphs from both source code and program binaries. First, assembly instructions are lifted into RISC-like expressions using VEX-IR [166] to obtain assignment formulas. Furthermore, the assignment formulas are simplified to S-Expressions by leveraging Z3[21] theorem prover [65]. Second, input/output behaviour of assignment formulas are sampled by using random concrete input values to capture the basic block semantics. Afterwards, MinHash [119] is used to reduce the complexity of similarity measurement amongst two basic blocks. Finally, in order to match the entire signature with a given target function, a greedy but locally-optimal graph matching algorithm called Best-Hit-Broadening (BHB) is proposed. BHB algorithm first performs basic block matching and further explores the immediate neighborhood nodes using Hungarian method [87] (no backtracking) to identify additional possible matches. However,

[21] https://github.com/Z3Prover/z3. Accessed on Dec 20, 2020.

the proposed approach is not practical at large scale, since k-MinHash degrades the performance quite significantly [174]. MULTI-MH is examined on the DD-WRT, NetGear, SerComm, and MikroTik firmware images.

A cross-architecture cross-OS binary search engine called BINGO [41] is proposed. First, a selective inlining technique, which inlines the callees into the caller functions is applied in order to cover more function semantics. Second, three types of filters, from fine-grained to coarse-grained, are leveraged in order to prune the search space. The first filter is OS dependent, while the second and third filters are cross-OS and cross-architecture. These filters are based on the identical library calls, operation types of library calls, and instruction types, where different weights are assigned to each filter. The top N dissimilar candidate functions based on the overall Jaccard distance [39] on three filters are discarded. For the purpose of function matching, first the effects of partial traces (tracelets) execution on the machine state (memory, general registers, and condition-code flags) based on symbolic expressions are extracted. Next, Z3 solver is used to generate Input/Output (I/O) samples and is further leveraged to eliminate infeasible partial traces as much as possible in order to shortlist candidate functions. Moreover, compiler-related partial traces are generalized into patterns and are removed from the partial traces in order to additionally prune the search space. Finally, a function is modelled with partial traces with various lengths, in which Jaccard containment similarity [1] is utilized to measure the final similarity scores. REIL [71] is used to lift up the assembly instructions to an intermediate representation. However, handling inlined functions which are invoked indirectly is not feasible. In addition, all floating-point instructions are not handled. Moreover, the accuracy of BINGO is affected by compiler optimizations [109].

A patch analysis framework named SPAIN [214] employs semantic analysis to identify the security patches and then summarizes the patch patterns through taint analysis [55]. First, similar functions in the original and patched versions are identified using BINDIFF.

Moreovder, for the sake of scalability, a 3D-CFG-based hashing technique [45] is used to filter out the functions with no changes or with minor compiler-related changes. Second, pair-wise basic block matching is employed to identify patched basic blocks, the relationship amongst their partial traces [41] and their relationship to the original function. Third, a semantic analysis by leveraging the effects of the partial traces execution [41] on machine state is performed to distinguish the security patches from non-security patches. Finally taint analysis is employed to summarize the vulnerability patterns. However, only some specific vulnerabilities namely, integer overflows, buffer overflows, and double-free/use-after-free (UAF) vulnerabilities are covered in this work. Additionally, SPAIN focuses solely on the vulnerabilities patched in one function.

A statistical approach named ESH[22] [60] decomposes the code into smaller fragments and then searches for similar binary functions deployed on different CPU architectures based on the similarity between functions fragments. ESH divides the function code into comparable fragments of *strand*, which are extracted from the basic block based on their data dependencies using def-use chain [209]. Then, the assembly code is lifted into Boogie intermediate verification language (IVL) [138] and then the *strands* are compared semantically through the BOOGIE[23] program verifier [25]. The verifier determines the equivalence between two strands through checking input-output equivalence. Furthermore, ESH proposes the Local Evidence of Similarity (LES) to amplify unique strands and give less significance to popular strands that are usually generated by compilers (e.g., prolog/epilog). Afterwards, strands similarities are lifted into function similarity via utilizing a statistical reasoning model. Therefore, the more semantically similar strands are found between two functions, the more likely these two functions are equivalent. However, utilizing the verifier is computation intensive, which prevents the proposed solution to be scalable. Additionally, the relation between basic blocks has not been taken into account.

[22] https://github.com/tech-srl/esh, http://binsim.com/. Accessed on Dec 20, 2020.
[23] https://github.com/boogie-org/boogie. Accessed on Dec 20, 2020.

Similarly, GITZ [61] searches for known vulnerable functions in a large corpus of binary functions cross-compiled for different CPU architectures with variant compiler optimization settings. First, GITZ extracts the comparable fragments of *strands*. Second, each fragment is lifted into VEX-IR in order to support multiple architectures, and then lifted again to LLVM intermediate instructions to utilize the re-optimization built-in *clang* feature. The *clang* re-optimization is utilized to build canonical representation and ultimately to ease the searching process by performing text comparisons. Each fragment gets a statistical ranking generated from statistical reasoning to distinguish the significant fragments from others that may cause false positive, such as fragments that handle stack or memory. Finally, the signature of each function is formulated as a set of MD5 hash values for every generated fragment. Finally, the two set of hashes are compared. However, according to the reported evaluation results, GITZ cannot detect *Heartbleed* vulnerability[24] accurately, with a false positive rate of 52%. Furthermore, it suffers from not handling various memory layouts over different CPU architectures.

Another approach called FIRMUP [62] is proposed to identify vulnerable functions in the firmware images. Similar to the previous work of the same group of authors, i.e., GITZ, first the functions are decomposed into basic blocks, and then slicing is applied on the basic blocks to obtain the strands. Then, compiler optimizer and normalizer are utilized to transfer the semantically equivalent strands into a canonical form (syntactic form). The more the functions share the same strands, the more they are similar. Additionally, to improve the accuracy of FIRMUP, a back-and-forth games algorithm [76], called Ehrenfeucht-Fraïssè, is leveraged to perform the matching for the neighboring functions and therefore to extend a more appropriate partial matching. FIRMUP is tested on about 2000 firmware images from various device vendors, including NetGear, D-Link and ASUS.

[24] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160. Accessed on Dec 20, 2020.

Another approach called XMATCH [83] searches for vulnerable functions in a cross-platform environment. The authors built *conditional formulas* from binary code functions (where the instructions are lifted to IR using McSema[25] [67]) as a higher level semantics. *Conditional formulas* are resilient to CFG structure changes as well as to the variation of CPU architectures instruction set. XMATCH utilizes both data dependencies and condition checks, to detect both absent or incorrect condition check, and erroneous data dependencies. In addition to capturing similar functions, XMATCH reduces the time needed to manually check the final candidate functions, since it provides in-depth information that helps in explaining the discovered vulnerabilities. However, in the case of function inline, the inter-procedure analysis should be employed to identify multiple functions, which is not supported yet. It is tested on the firmware image of the Linux-based DD-WRT router.

2.3.3 Distance-Based Approaches

The distance-based approaches extract different sets of features for a function, and then various similarity detection approaches are applied on the selected features in order to find the final set of candidates.

A binary search engine called RENDEZVOUS [125] extracts multiple features including *n-grams, n-perms*, *mnemonics*, *control flow subgraphs* and data *constants* to form the tokens. The *Nauty* graph library [155] is used to convert the subgraphs into canonical forms. The disassembled functions are broken down into short tokens, and a probability is assigned to their occurrence in the reference corpus, based on a statistical model. Two query models of set-based Boolean model (BM) and a distance-based vector space model (VSM) are used to filter and then score the document queries, respectively. The indexing and querying are performed by leveraging *CLucene*[26] text search engine and the *Bloom filter* [29] is

[25] https://github.com/trailofbits/mcsema. Accessed on Dec 20, 2020.

[26] http://clucene.sourceforge.net, https://github.com/synhershko/clucene. Accessed on Dec 20, 2020.

utilized to select unique terms. However, matching control flow subgraphs is performed regardless of the basic block instructions, which results into a high false positives rate.

A cross-architecture binary code similarity approach called GEMINI[27] [213] is proposed based on a neural network model. It first extracts the control flow graphs attributed with manually selected features called attributed control flow graphs (ACFG). Subsequently, it then employs `structure2vec` [59] combined with *Siamese* architecture [31] to generate the graph embeddings of two similar functions close to each other. Introducing embedding with deep learning by GEMINI highly improves binary function fingerprinting over multi-platform CPU architectures. However, the embedding generated by GEMINI relies mainly on the statistical features without considering the relationships between them and the instruction sets represented by these features. The reported vulnerability identification accuracy of about 82% reflects the limitation of such feature choices to be applied to this problem. Similar to GENIUS, $8,128$ firmware images from 26 vendors with different products, such as IP cameras, routers, and access points are indexed in their repository.

Similarly, VULSEEKER[28] [91] first extracts the *labelled semantic flow graph* (LSFG), which are obtained by combining the CFG with DFG, and then propose a semantics-aware deep neural network model in order to generate the function embeddings. Finally, the *cosine* similarity is used to measure the similarity between two functions. VULSEEKER is examined on $4,643$ cross-architecture firmware images.

An approach called, αDIFF [142], extracts function code (raw bytes), function calls and imported functions. The convolutional neural network (CNN) and a *Siamese* network are used to convert the function code into embeddings. Finally, the three intra-function, inter-function and inter-module distances from a given function to the vulnerable functions in the repository are measured. Similar to most of the previously mentioned works, αDIFF is evaluated on DD-WRT, and NetGear ReadyNAS firmware images.

[27] `https://github.com/xiaojunxu/dnn-binary-code-similarity`. Accessed on Dec 20, 2020.
[28] `https://github.com/buptsseGJ/VulSeeker`. Accessed on Dec 20, 2020.

Inspired by self-attentive neural networks [141], SAFE[29] [151] learns function embedding automatically from the function instructions. Each assembly instruction is considered as a *word* and the sequence of assembly instructions is considered as a *sentence* in natural language. The `skip-gram` [159] is utilized to convert the assembly instructions into vectors. Then, GRU recurrent neural network (GRU RNN) [94] is employed to capture the relationship between sequential assembly instructions, which are presented in the form of vectors. The use of attention mechanism helps the model assign higher weights to more informative parts of the code. SAFE utilizes the `skip-gram` method to convert the assembly instructions into vectors. Finally, a *Siamese* neural network composed of two self-attentive neural networks is employed to train the final model.

More granular than looking for similar binary function, INNEREYE [224] introduces Neural Machine Translation (NMT) to determine whether there is a similarity between a given basic block and another piece of code compiled for a different CPU architecture. It considers assembly instruction with its operands as a single *word* and the whole basic block as a *sentence*. First, assembly instructions are converted into embedding using `skip-gram with negative sampling` (SGNS) [159] model to capture the contextual meaning of the word. To compare the semantic similarity of two basic blocks on different architectures, INNEREYE treats similar basic block cross-compiled for different CPU architectures similar to handling equivalent sentences written in different natural languages. To this end, *Siamese* neural network composed of two identical long short-term memory (LSTM) [94] models is employed. The dataset and the neural network models are publicly available[30].

An unsupervised feature learning model called ASM2VEC[31] [69] is proposed, which learns latent representation of assembly instructions of a function and finally represents

[29] `https://github.com/gadiluna/SAFE`. Accessed Dec 20, 2020.
[30] `https://nmt4binaries.github.io/`. Accessed on Dec 20, 2020.
[31] `https://github.com/McGill-DMaS/Kam1n0-Community`. Accessed on Dec 20, 2020.

each binary function as a vector. ASM2VEC utilizes natural language processing (NLP) learning model inspired by PV-DM model [136], which is used to automatically learn documents representation. ASM2VEC models the control flow graph of each function using a sequence of execution traces by employing *random walks* and *edge coverage*. Moreover, inspired by BINGO, selective function inlining is proposed. The concatenation of these features is considered as function representation. Eventually, the learned latent function representation is used to compare with other vectors indexed in the repository by using *cosine similarity* function to retrieve the top-k candidate functions. However, ASM2VEC is designed for a single CPU architecture, i.e., x86.

2.3.4 Symbolic Execution Combined with Static Analysis

There exist some works that combine static analysis with symbolic execution. In order to compare two programs, BINHUNT [90] employs maximum common subgraph (MCS) isomorphism using backtracking algorithm [128, 204]. The model is applied on the intermediate representation of x86 assembly instructions in order to compare both the CFGs and call graphs of two binary programs. Additionally, symbolic execution combined with simple theorem proving (STP) [89] is used to compare two basic blocks.

The code reuse detection approach called EXPOSÉ [169] combines semantic execution (using STP constraint solver) with the syntactic matching (using *cosine* distance of the function *n-grams*). The BAP [34] framework is modified to process x86 instructions in order to be used with the STP. A filtering process is used to prune the search domain by both excluding compiler loader support functions and identifying improbable function pairs based on four attributes (number of input arguments, number of out-degrees, function size, and cyclomatic complexity). However, in order to identify the vulnerable functions EXPOSÉ is too coarse-grained in some cases. The main limitation of this work is that semantic execution is too strict while syntactic matching, on the other hand, is too coarse.

An obfuscation-resilient method named CoP [145, 146] combines symbolic execution with longest common subsequence (LCS) algorithm to prform function matching in program binaries. The basic block semantics is modelled by symbolic formulas, which are obtained from the input-output relations of the block. Then, the similarity of two execution paths using LCS with basic blocks as elements is modelled. However, the computational overhead of symbolic execution is high, which is not practical at large scale.

A cross-architecture framework called FIRMALICE [192] investigates the existence of various authentication bypass vulnerabilities commonly known as *backdoors* in firmware images. It declares that any execution path derived from the entry point of the firmware to a privileged operation should go through a solid input validation process. Hence, an attacker cannot bypass by means of information retrieval from the firmware image itself. To this end, FIRMALICE initially utilizes static analysis to extract the program data dependency graph, and then extracts the program slices leading from the entry point to privileged operation location determined by a security analyst. Then, it employs its symbolic execution engine, inspired by KLEE [36], MAYHEM [40], and FUZZBALL [17] to find possible successful paths that lead to the desired privileged location.

2.3.5 Comparative Study

A comprehensive evaluation of the existing works is not feasible, mainly due to the absence of their dataset. However, we conduct a qualitative comparison based on the information provided by each solution in terms of the approaches, implementations and evaluations. The findings of this study are summarized in Table 2.1 and Table 2.2. The first and second columns of Table 2.1 specify the proposals and the corresponding venues ordered by the date. Columns three to six present the features used by each proposal, based on our proposed features taxonomy presented in Figure 2.1. The next four columns indicate the types of analysis based on our taxonomy presented in Figure 2.3. In the next column, we

provide the corresponding main methodologies. The next two columns provide the disassembler and the use of existing frameworks as well as use of intermediate representation. Afterwards, the "Mapping Results" column indicates the in-depth analysis of the output. A bullet mark indicates that a work provides the matching results (e.g., corresponding matched instruction sets or basic blocks) in addition to a final similarity score. Finally, the last two columns show which tools are open source and which ones are accessible to the public. Furthermore, we provide the distribution of different features (e.g., types of analysis) among different surveyed works, as shown in the last row. For instance, 68% of the

PROPOSAL	VENUE	Feature(s)				Analysis				Methodology	Disassembeler/ Reused Framewrok	IR		Release	
		Instruction-level	Semantic	Structural	Statistical	Symbolic Execution	Graph-based	Data Flow-based	Distance-based				Mapping Results	Open Source	Open Service
BINHUNT [90]	ICICS'08			•		•	•			MCS	IDA, BAP	BAP			
BINHASH [119]	ICMLA'12		•					•		ED, LSH	IDA				
EXPOSE´ [169]	COMPSAC'13			•		•			•	ED	IDA, BAP				
BINJUICE [132]	PPREW'13		•				•			ED	Objdump				
RENDEZVOUS [125]	MSR'13	•		•					•	ED	DYNIST		•		
COP [146]	FSE'14	•			•	•	•	•		LCS	IDA, BAP	BAP			
TRACY [63]	SIGPLAN'14		•	•			•			LCS, DFA	IDA			•	
TEDEM [175]	ACSAC'14		•	•			•			GED	IDA, BINDIFF	METASM [98]			
SIGMA [10]	DFRWS'15		•	•			•			GED	IDA				
MULTI-MH [174]	S&P'15		•	•				•		BHB, MinHash	IDA	VEX [166]			
BINGO [41]	FSE'16		•						•	JD	IDA	REIL [71]			
LIBV [177]	SE'16	•		•				•		DFA, GM	IDA				•
KAM1N0 [68]	KDD'16	•		•			•			GM, LSH	IDA			•	•
ESH [60]	SIGPLAN'16		•	•				•		DFA, SR	IDA, BAP	LLVM, BoogieIVL			•
DISCOVRE [78]	NDSS'16			•	•		•			MCS, JD	IDA				
GENIUS [84]	CCS'16			•	•		•			LSH, JD	IDA				
SPAIN [214]	ICSE'17		•					•		JD	IDA				
GITZ [61]	PLDI'17		•	•				•		DFA,SR	ANGR	VEX, LLVM			
BINSEQUENCE [110]	ASIACCS'17	•	•	•			•			LCS, LSH, GM	IDA		•		
GEMINI [213]	CCS'17				•				•	DNN	IDA			•	
XMATCH [83]	ASIACCS'17		•	•				•		DFA,GED	IDA	McSema [67]	•		
FIRMUP [62]	ASPLOS'18		•					•		DFA, SR	IDA	VEX, LLVM			
FOSSIL [11]	TOPS'18		•	•	•		•			ED	IDA				
VULSEEKER [91]	ASE'18		•	•					•	DNN	IDA				
αDIFF [142]	ASE'18	•		•					•	DNN	IDA				
ASM2VEC [69]	S&P'19		•	•					•	NLP	IDA			•	
INNEREYE [224]	NDSS'19		•						•	NLP, LCS	IDA				
SAFE [151]	DIMVA'19		•						•	NLP	IDA, ANGR,RADAR2			•	
DISTRIBUTION	NA	21%	64%	68%	18%	11%	39%	32%	32%	NA	NA	32%	11%	18%	11%

(•) means that the approach provides the corresponding feature, it is empty otherwise. (BHB) Best-Hit-Broadening, (DNN) Deep Neural Network, (DFA) Data Flow Analysis, (ED) Distance-based (e.g., Euclidean), (GED) Graph Edit Distance, (GM) Graph matching, (LSH) Locality Sensitive Hashing, (JD) Jaccard Distance, (LCS) Longest Common Subsequence, (MCS) Maximum Common Subgraph Isomorphism, (NLP) Natural Language Processing, (SR) Statistical Reasoning. The 'DISTRIBUTION' presents the percentage of each category used in the selected proposals. For instance, 68% of the exiting solutions rely on structural features, while only 18% extract statistical features. The grey cells are for the sake of readability to separate different categories.

Table 2.1: A comparison of state-of-the-art static approaches

exiting solutions rely on structural features, while only 18% extract statistical features. The gray shadings are used only for visualization purpose, in order to group related features.

PROPOSAL	Programming Lan.	Dataset		Compiler(s)				CPU Arch.			OS		Detection			Normalization	Accuracy	Performance	Comparison	Filtering	Obfuscation
		Normal Binary	Firmware	VS	GCC	ICC	Clang	x86-64	ARM	MIPS	Window	Linux	Known Vul.	Unknown Vul.	Normal Function						
BINHUNT [90]	–	6	NA	●	●			●			●	●	●		●		●	●	0		
BINHASH [119]	–	16	NA	–	–	–	–	●			●				●	●			0		
EXPOSE´ [169]	–	3,075	NA	●				●				●			●		●	●	0	●	
BINJUICE [132]	Prolog, Python	70	NA		●			●			●	●			●				0		
RENDEZVOUS [125]	C++	98	NA		●		●	●				●			●	●	●	●	0		
COP [146]	C++	321	NA		●	●		●				●					●		4		
TRACY [63]	Python	–	NA		●			●				●	●			●	●	●	2		
TEDEM [175]	C++	15	NA	●				●			●	●	●		●		●	●	0	●	
SIGMA [10]	–	18	NA	●	●			●			●				●	●	●		0		
MULTI-MH [174]	C++	60	4		●		●	●	●	●	●	●	●				●	●	0		
BINGO [41]	Python	110	NA		●		●	●	●		●	●	●	●		●	●	●	4	●	
LIBV [177]	–	9	NA	●				●			●				●	●	●	●	1	●	
KAM1N0 [68]	Java, Python	10	NA	–	–	–	–	●			●				●	●	●	●	4		
ESH [60]	C#, Python	1,000	NA		●	●	●	●				●	●				●	●	2		
DISCOVRE [78]	–	2,280	2	●	●	●	●	●	●	●	●	●	●				●	●	2	●	
GENIUS [84]	Python	17,626	8,128		●		●	●	●	●		●	●				●	●	3	●	
SPAIN [214]	–	28	NA	–	–	–	–	●			●	●	●			●	●	●	0	●	
GITZ [61]	–	–	NA		●	●	●	●	●			●	●				●	●	0		
BINSEQUENCE [110]	C++, Python	19	NA	●				●			●		●			●	●	●	4	●	
GEMINI [213]	Python	51,314	8126		●			●	●	●	●	●	●				●	●	2		
XMATCH [83]	–	72	1		●		●	●		●	●	●	●				●	●	4		
FIRMUP [62]	–	200,000	2,000	–	–	–	–	●	●	●		●	●			●	●	●	2		
FOSSIL [11]	Python	6925	NA	●	●	●	●	●			●	●			●	●	●	●	7		
VULSEEKER [91]	Python	–	4,643		●			●	●	●		●	●				●	●	1		
αDIFF [142]	–	67,427	2		●		●	●	●	●		●	●				●		6		
ASM2VEC [69]	Java, Python	1116	NA		●	●	●	●				●	●			●	●	●	12		●
INNEREYE [224]	Python	5	NA				●	●	●						●	●	●	●	1		
SAFE [151]	Python	11	NA		●		●	●	●			●	●				●	●	1		
DISTRIBUTION	NA	82%	29%	29%	68%	21%	46%	100%	39%	29%	54%	78%	64%	4%	39%	46%	86%	82%	61%	29%	4%

(●) means that the approach provides the corresponding feature, it is empty otherwise. (–) means that the information is not provided or partially provided. (NA) means that the corresponding work does not consider firmware analysis. The 'DISTRIBUTION' presents the percentage of each category used in the selected proposals. For instance, 100% of the exiting solutions support x86 architecture, while only 29% support MIPS architecture. The grey cells are for the sake of readability to separate different categories.

Table 2.2: A comparison of state-of-the-art static analysis implementations and evaluations

We further conduct a comparative study on the implementation and evaluation of existing works as listed in Table 2.2. The first column lists the proposals. The second column presents the used programming languages in each proposal. The next two columns indicate the dataset (normal binary and firmware) used for the experiments. The next four columns mark the employed compilers utilized to prepare the ground truth. The next three columns mark the CPU architectures that are supported by these approaches. In the next

two columns, the operating systems is marked. Afterwards, the "Detection" criteria provides the type of output. It is marked when a work identifies known vulnerable functions, unknown vulnerabilities or it performs normal function matching. The last three columns show works that use normalization and provide accuracy and performance results. Next column indicates the number of works which have been compared with the current work. The last two column indicate whether the work is using any filtering process, and if it supports obfuscated binaries, respectively. We also provide the distribution of these features over all proposed works (e.g., the ratio of the works that support different CPU architectures). For instance, 100% of the exiting solutions support x86 architecture, while only 29% support MIPS architecture.

Discussion. The key observations of this comparative study are as follows: First, there exist several features utilized in the literature, which are shown to significantly improve the efficiency and accuracy of the vulnerability detection solutions. As can be observed, semantic and structural features are the most frequently used features. Second, there is no single best solution to identify vulnerable functions. Amongst the employed solutions, graph-based approaches combined with data flow-based approaches demonstrate the best practice to be chosen for the vulnerability detection. More recently, distance-based approaches which employ deep neural networks (DNN) and natural language processing (NLP) show the best results for cross-architecture vulnerability detection. Third, the filtering process is a promising solution to overcome the scalability issue. However, filtering approaches should be carefully designed and thoroughly evaluated to assure the accuracy. Fourth, to the best of our knowledge, there exists only one work, namely BINGO [41], that identifies unknown vulnerabilities in normal binaries. Finally, even though MinHashing and LSH are employed for function matching, existing works under this category are not practical at large scale due to their time complexity for functions with large and complex control flow graph. To conclude, most of the recent static solutions employ graph-based approaches on

x86 architecture for binaries compiled with the *GCC* compiler on Linux platform. Moreover, the presented comparison highlights the trend of binary analysis, which is moving towards DNN and NLP techniques on Linux platform to overcome cross-architecture problem and the scalability issue of online searching.

As seen, the existing approaches can overcome some of the challenges, such as those related to compiler effects (C2) and hardware architecture (C5). However, some challenges still remain unresolved. In the following, we discuss the fundamental limitations of the static approaches.

Detecting unknown vulnerabilities. Most of the static solutions define a pattern/signature for a function, and then perform function matching. Therefore, the signatures of already known vulnerable functions are stored in the repository such that the vulnerable functions can be discovered by identifying any match within the signatures in the repository. However, there might be some functions with unknown vulnerabilities, which could not be identified in this manner (C6). Unknown vulnerabilities might be identified by employing data flow and dynamic analysis.

Detecting run-time vulnerabilities. Static approaches fail to detect vulnerabilities that are exploited during the execution time (C6). For instance, the run-time data-oriented exploits cannot be detected due to the lack of execution semantics checking [47]. A similar situation is encountered in the case of network activities, since this information will be provided during the runtime process.

Identifying inline functions. Function inlining (C4) may introduce additional complexity to the vulnerable function detection problem, since it requires to fingerprint a function containing partial code from another function. Static approaches generally fail to identify inline functions. However, data flow analysis and symbolic execution could be employed as potential solutions to this problem. Systematically addressing this problem is still an open challenge.

Scalability using filtering. The scalability issue (C9) of static analyses approaches has been somewhat addressed by filtering processes. During such processes, the high-likely dissimilar or non relevant functions will be excluded from the analysis. Therefore, filtering processes minimize the search space in order to statically identify the vulnerabilities more efficiently, and also to provide a better code coverage in the case of dynamic analysis. However, the filtering process may affect the accuracy. Therefore, examining the filtering processes could help better learn the pros and cons of each corresponding method, and further propose new efficient and accurate filtering techniques.

Lack of semantic insights and replaying vulnerabilities. Static approaches provide a list of potential vulnerable functions with relatively high false positives rates (C6). Therefore, manual effort is required in order to verify the obtained results. These techniques do not provide any information on how to trigger the discovered vulnerabilities for further investigation and to replay the attacks (C7). Therefore, other approaches (e.g., symbolic execution) could be employed to produce repayable inputs in order to validate the vulnerabilities and further provide semantic insight on the reason of the execution and the corresponding part of the code. On the other hand, static approaches could be employed to overcome the scalability issue of pure dynamic analysis and symbolic execution techniques.

Generalizing vulnerability signatures. Most of the existing approaches provide a specific pattern in different representations and semantic levels for each vulnerable function, and then employ a matching technique or a similarity measurement to identify it. Providing a general signature for each vulnerability (e.g., buffer overflow) rather than matching with the functions that already have a specific vulnerability is one of the future directions.

2.4 Summary

This chapter reviews the literature in binary analysis with emphasis on static solutions. Additionally, we devise a taxonomy of different types of utilized features, application domains and analysis techniques followed by a qualitative comparison. To tackle the drawbacks of existing approaches, this thesis provides the design of a framework that is composed of four major components: (i) BINCOMP: identifying compiler and compiler-related functions; (ii) BINSHAPE: determining standard library functions; (iii) BINARM: discovering vulnerable functions in firmware images of intelligent electronic devices in the smart grid; and (iv) TIOHTIÀ:KE: preforming code similarity detection in cross-compiled cross-architecture obfuscated binaries and identifying vulnerable functions.

Chapter 3

Compiler Provenance Attribution

Compiler identification is an essential component of binary toolchain analysis and malware analysis. Security investigators are often tasked with the analysis and attribution of malicious binary code, which needs to be done quickly and reliably. Such binaries can be a source of intelligence on adversary tactics, techniques, and procedures. Compiler provenance information can aid binary analysis by uncovering fingerprints of the development environment and related libraries, leading to accelerated analysis. In this chapter, we present BINCOMP, which is a practical multi-layered approach for analyzing the syntax, structure, and semantics of disassembled functions to extract compiler provenance.

This chapter is organized as follows. The compiler provenance problem and an overview of our approach are introduced in Section 3.1. Section 3.2 provides the presented features that are employed in BINCOMP. Section 3.3 and Section 3.4 present two different approaches (BINCOMP and ECP) for the compiler identification problem. The evaluation results of BINCOMP are presented in Section 3.5. Then, Section 3.6 discusses its limitations before drawing the conclusions and hinting on future research directions.

3.1 Introduction

Program binaries often lack meaningful identifiers, function names and code comments; all of which would have been relevant to understanding the context of program behavior. Furthermore, variations in compiler front-end languages, back-end transformation logic, and optimization heuristics complicate matters further. Compiler provenance can reveal information regarding the behavior, family, type, version, optimization, and functions of the originating compilers [19].

The primary research on compiler provenance encompasses the core studies of source compiler identification [187], function labelling [114], and toolchain recovery [184]. Certain limitations can be identified in the existing solutions. These include the absence of meaningful information about compilers, the use of computationally-intensive feature ranking techniques, and extensive training set requirements. Such limitations could impact the practical application of such methods. More specifically, a malware detection approach [202] analyzes the instruction frequencies produced by a specific compiler to determine whether a program has been compiled with a know compiler or it is a malware. However, this approach does not extract information regarding the compiler property (e.g., version and optimization level). Furthermore, existing tools which employ heuristics (e.g., IDA PRO[1], PEID[2], RDG[3]) are capable to identify commonly known compilers, assuming the availability of type signatures. However, the usage of generic and rigid signatures and applying exact matching algorithms may fail, for instance when a slight difference between the signatures is present.

Other existing techniques (e.g., [184, 187]) rely on generic signatures in conjunction with compiler attribute feature ranking and predefined templates. This usually leads to a large amount of irrelevant features and consequently results in additional time consumption

[1] https://www.hex-rays.com/products/ida/. Accessed on Dec 20, 2020.
[2] https://github.com/wolfram77web/app-peid. Accessed on Dec 20, 2020.
[3] http://www.rdgsoft.net/. Accessed on Dec 20, 2020.

and computational complexity. Furthermore, the top-ranked features obtained as a result of feature selection may not describe compiler-specific functions, from a semantic or a structural perspectives. However, filtering compiler-related functions is a critical pre-processing step for reducing false positive rates in binary analysis. Even so, existing approaches (e.g., [184, 187]) do not consider compiler-related functions identification.

This chapter extends the existing approaches by applying new heuristics in feature selection and processing based on domain-specific knowledge. This leads to reduction in the computational complexity of the search processes and improvements in the accuracy of the detection process by leveraging more stable aspects of compiler behavior instead of byte-level classification (e.g., [187]). Additionally, the proposed approach combines various engineered feature categories to capture structural, syntactic, and semantic aspects of compilers in support of provenance elicitation. Moreover, it facilitates function analysis via identification of compiler/linker-dependant functions.

The objective of this chapter is to provide a practical framework for characterizing compiler behavior using a set of discriminative feature profiles to facilitate compiler comparability, identification and property measurement. Through experiments with supervised compilation involving multiple toolchains, we observe that certain compiler utility and helper functions are steadily preserved during the compilation and linkage processes, across distinct program contexts. Such behavior can be determined by inspecting the code generation and emission mechanisms of source compilers[4] as well as the linked/runtime libraries. We assume that this property holds true for compilers/linkers in the scope of our analysis. Our approach pivots on such functions along with other compiler/linker characteristics to build plausible support for the most likely compiler of target binaries.

Contributions. The contributions of this chapter are as follows:

- We introduce BINCOMP, a practical approach based upon techniques of function fingerprinting that enable the identification and recognition of compiler families,

[4] https://gcc.gnu.org/onlinedocs/gccint/Machine-Desc.html. Accessed on Dec 20, 2020.

helper/utility functions, optimization levels and versions within a unified framework.

- The proposed recognition patterns are based on stable features of compiler behavioral profiles, as opposed to file-level properties (e.g., header information) which are primarily used by existing detection tools. Such information may be unavailable in stripped binaries. Our method captures structural, syntactical, and semantic aspects of compiler behavior.

- We apply new heuristics to feature selection and the processing methods based on domain-specific knowledge, resulting in reduced computational overhead and increased sensitivity during the detection process. The adopted features facilitate compiler comparability and property measurement.

- The proposed provenance methodology is evaluated on a large collection of program binaries compiled with different compilers and compilation settings. The performed experiments demonstrate that BINCOMP achieves from 86% to 90% accuracy in compiler family identification, and 90% accuracy in optimization level detection.

3.1.1 Motivating Example

In this subsection, we present our motivation uisng an example. To compare compiler generated code under various compilation toolchains, we consider a simple C++ program as shown is Listing 3.1 and then we compile it with three different compilers. This simple program contains a `main` function that defines an integer variable `num` and assigns it the value of 11. Then, a stream I/O library call is made to print a string, followed by an increment in the variable value. The program then terminates by returning the integer value of `num`.

We compile the program with MSVCP14 and disassemble the obtained binary with IDA PRO (as presented in Listing 3.2). We obtain 90 assembly functions from the code

segment, of which, 13 are library functions, 76 are related to compiler utility/helper functions and a single user function. When we compile the program with MSVC 2010, this program results in 37 assembly functions in fully optimized release mode. The disassembly includes 31 compiler utility/helper functions, five library, and one user function. Compilation and disassembly of the same program with MSVC 2012 generates 72 functions (with security checks enabled) of which 51 functions are related to the compiler, 18 are library functions, and three are user functions. While comparing the compiler-related functions, we notice that 25 functions remain the same. In addition, we conduct the same simple experiment with a slightly more complex programs, and we find that the same set of 25 compiler-related functions are present in those disassembled programs.

Listing 3.1: A Simple C++ Program

```
#include <iostream>
using namespace std;

int main() {
  int num = 110;
  cout << "hnum=" << num+1;
  num++;
  return num;
}
```

Moreover, Listings 3.2, 3.3, and 3.4 show the disassembled versions of the simple program in Listing 3.1 compiled with MSVCP (Windows), *GCC* (Linux), and *clang* compilers, respectively, using IDA PRO and OBJDUMP disassemblers. As can be seen, the compilers generate different code sequences for an identical input. Consequently, we also build compiler-specific behavioral profiles based on code transformations. These profiles describe compiler characteristics with respect to language structure transformation, source to assembly mappings, and code optimizations.

Listing 3.2: Compiled with MSVCP 14 on Windows, disassembled with IDA

```
var_4 = dword ptr -4
argc = dword ptr 8
argv = dword ptr 0Ch
envp = dword ptr 10h

push ebp
mov ebp, esp
push ecx
mov [ebp+var_4], 6Eh
mov eax, [ebp+var_4]
add eax, 1
push eax
push offset Str ; "hnum="
mov ecx, ds:std::basic_ostream<char,std::char_traits<char>> std::cout
push ecx ; int
call std::operator<<<std::char_traits<char>>(std::basic_ostream<char,
std::char_traits<char>> &,char const *)
add esp, 8
mov ecx, eax
call ds:std::basic_ostream<char,std::char_traits<char>>::operator<<(int)
mov edx, [ebp+var_4]
add edx, 1
mov [ebp+var_4], edx
mov eax, [ebp+var_4]
mov esp, ebp
pop ebp
retn
```

Listing 3.3: Compiled with *GCC* 8.1, disassembled with OBJDUMP

```
push rbp
mov rbp,rsp
sub rsp,0x10
mov DWORD PTR [rbp-0x4],0x6e
mov esi,0x400825
mov edi,0x601060
call 4005f0 <std::basic_ostream<char, std::char_traits<char> >&
std::operator<< <std::char_traits<char>>(std::basic_ostream<char,
std::char_traits<char> >&, char const*)@plt>
mov rdx,rax
mov eax,DWORD PTR [rbp-0x4]
add eax,0x1
mov esi,eax
mov rdi,rdx
call 400610 <std::ostream::operator<<(int)@plt>
add DWORD PTR [rbp-0x4],0x1
mov eax,DWORD PTR [rbp-0x4]
leave
ret
```

Given a list of disassembled functions, we aim at identifying the most likely build toolchain, which the input binary was compiled/linked with. Considering different program compilations, we observe that a subset of compiler-related functions remains intact in all variations. To extract the list of such functions, we intersect multiple sets of compiled programs and develop signature profiles. However, we do not rely on string-based function identifiers as the only feature for comparison, since in stripped binaries and obfuscated code, such strings may not be available. Also, disassemblers often assign generic names to functions. Thus, we create numerical and symbolic representations of compiler/helper functions using multiple feature vectors as part of the compiler profiles.

Listing 3.4: Compiled with *clang* 7.0, disassembled with OBJDUMP

```
push rbp
mov rbp,rsp
sub rsp,0x10
mov DWORD PTR [rbp-0x4],0x0
mov DWORD PTR [rbp-0x8],0x6e
movabs rdi,0x601060
movabs rsi,0x400834
call 4005c0 <std::basic_ostream<char, std::char_traits<char>>&
std::operator<< <std::char_traits<char> >(std::basic_ostream<char,
std::char_traits<char> >&, char const*)@plt>
mov ecx,DWORD PTR [rbp-0x8]
add ecx,0x1
mov rdi,rax
mov esi,ecx
call 4005e0 <std::ostream::operator<<(int)@plt>
mov ecx,DWORD PTR [rbp-0x8]
add ecx,0x1
mov DWORD PTR [rbp-0x8],ecx
mov ecx,DWORD PTR [rbp-0x8]
mov QWORD PTR [rbp-0x10],rax
mov eax,ecx
add rsp,0x10
pop rbp
ret
```

3.1.2 Approach Overview

We establish a multi-layered architecture for BINCOMP comprising of three main processes, each aiming at capturing a different set of compiler behavior properties. Collectively, these processes enable the proposed solution to derive an accurate representation of source compiler behavior using syntactic, structural, and semantic profiles. Each process/layer builds upon a unique detection technique. The architecture of BINCOMP is depicted in Figure 3.1, which is composed of three layers:

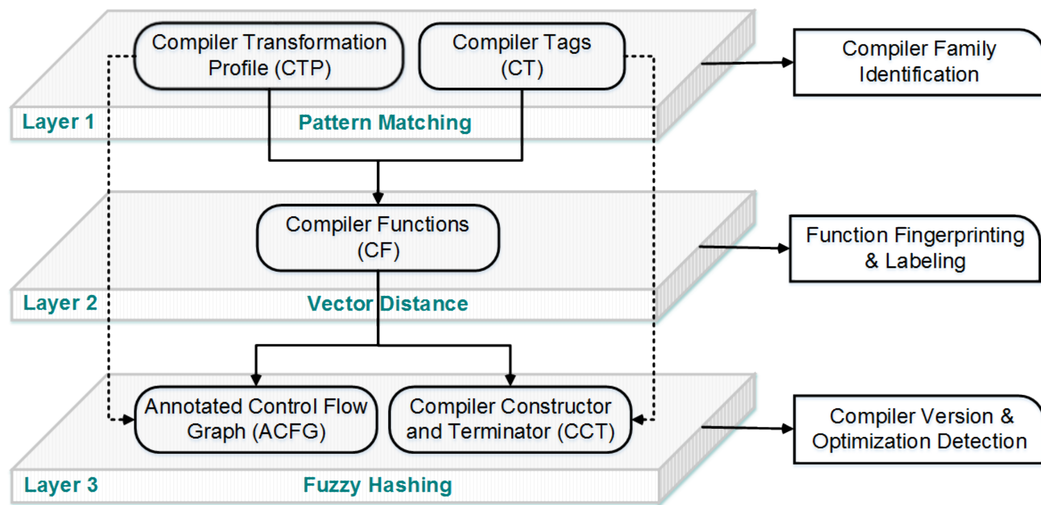


Figure 3.1: BINCOMP multi-layered architecture

- The first layer extracts syntactical features and generates two outputs, namely the Compiler Transformation Profile (`CTP`) and Compiler Tags (`CT`). The detection method is based on pattern matching against predefined/incremental signatures.
- The subsequent layer then identifies Compiler Functions (`CF`) by analyzing instruction-level features and building function profiles composed of numerical and symbolic feature vectors. This process aims to label helper and utility functions by measuring the similarity (calculating the distance) between known and target numerical vectors.
- The last layer extracts semantic features, which are captured using the Annotated Control Flow Graph (`ACFG`) and the Compiler Constructor and Terminator (`CCT`) profiles. These semantic profiles are evaluated in order to infer the optimization heuristics, level, and version of the compiler in conjunction with the helper/utility feature vectors. Inexact matching using fuzzy hashing is performed on semantic graphs extracted during this phase.

The proposed architecture can be viewed as the continuous integration and learning of compiler code conversion profiles with a prior knowledge of their code transformations.

This serves to adjust source compiler features as well as to update them when compiler-related functions undergo changes due to new versions or updates. Moreover, unique obtained behavorial profiles can be utilized to describe different characteristics of the binary code. This includes the sequence of compiler-specific function calls, function/library inter-dependencies, assembly instruction categories, and the control flow graphs of initialization and terminator utility/helper code. Our experimental results indicate that the solution is practical and effective in identifying the compiler of target binaries, and that the detection method is computationally efficient.

3.2 Feature Extraction

In this section, we introduce our features and provide more detailed information.

3.2.1 Compiler Transformation Profile

The Compiler Transformation Profile (`CTP`) feature captures syntactical aspects of compiler behavior. This feature describes how compilers reflect source-level code and data abstractions (e.g., control statements, object, queue, stack, list, array, calls, etc.) into assembly-level instructions. For example, a simple `if-then-else` statement can be represented as `cmp|test` then `jcc` (conditional jump) in MSVC binaries. This feature is obtained as the outcome of a supervised compilation process explained in Section 3.3.1.

Example 3.2.1 *The output of three compilers given an identical input (as shown in Listing 3.5) is presented in Figure 3.2. The excerpt shows a C++* `switch-case` *statement compiled with (1) MSVC, (2) GCC and (3) clang compilers, respectively. Despite their similarities, it is evident that each compiler family makes a specific choice on the selection and sequence of assembly mnemonics, operands, registers, memory access, and data transfers.*

Such variations allow to extract identifiable features and to generate signatures for each compiler family.

Listing 3.5: Simple `switch-case` statement

```
char g;
cin >> g;
switch (g) {
      case 'A':
            cout << "A";
            break;
      case 'B':
            cout << "B";
            break;
      default:
            cout << "?";
            break;
}
```

In order to facilitate effective `CTP` signature generation, a subset of assembly features should be selected, which most appropriately captures the original code semantics. To this end, we extract binary features F_B from the normalized versions of assembly instructions of a function, and then perform statistical frequency analysis on the compiler generated code to obtain code transformation patterns for different functionalities (e.g., `if` statement). Finally, we extract signature Sig_{c_i} from the normalized versions of assembly instructions compiled with c_i compiler.

Instruction Normalization. Each assembly instruction contains two parts: the *mnemonic* and the *operands*. First, we classify the instruction mnemonics into 11 groups according to their operation semantics. For instance, mnemonics that perform logical operations, such as `xor`, `shl`, `ror`, are replaced with `LGC` tag indicating the `Logical` operations group. Therefore, the mnemonic identifiers are compared with a platform-specific group of instructions, and the identifier member is replaced with the respective group label. Similarly, to distinguish between general-purpose registers `reg`, direct memory references `mem`,

Switch Statement Transformations

ASM code compiled with MSVC

```
lea eax, DWORD PTR _g$[ebp]
push eax
mov ecx, DWORD PTR __imp_cin
push ecx
call std::char_traits<char>
add esp, 8

mov dl, BYTE PTR _g$[ebp]
mov BYTE PTR tv67[ebp], dl
cmp BYTE PTR tv67[ebp], 65
je SHORT $LN4@main
cmp BYTE PTR tv67[ebp], 66
je SHORT $LN5@main
jmp SHORT $LN6@main

$LN4@main:
push OFFSET $SG29712
mov eax, DWORD PTR _imp_cout
push eax
call std::char_traits<char>
add esp, 8
jmp SHORT $LN2@main

$LN5@main:
push OFFSET $SG29714
mov ecx, DWORD PTR _imp_cout
push ecx
call std::char_traits<char>
add esp, 8
jmp SHORT $LN2@main

$LN6@main:
push OFFSET $SG29716
mov edx, DWORD PTR _imp_cout
push edx
call std::char_traits<char>
add esp, 8
$LN2@main:
```

ASM code compiled with GCC

```
lea rax, [rbp-0x1]
mov rsi, rax
mov edi, 0x601180
call 400670 std::char_traits

movzx eax, BYTEPTR [rbp-0x1]
movsx eax, al
cmp eax, 0x41
je4007b3 <main+0x2c>
cmp eax, 0x42
je 4007c4 <main+0x3d>
jmp 4007d5 <main+0x4e>

mov esi, 0x4008c5
mov edi, 0x601060
call 400660 std::char_traits

jmp 4007e5 <main+0x5e>

mov esi, 0x4008c7
mov edi, 0x601060
call 400660 std::char_traits

jmp 4007e5 <main+0x5e>

mov esi, 0x4008c9
mov edi, 0x601060
call 400660 std::char_traits

nop
```

ASM code compiled with Clang

```
mov eax, 0x601180
mov edi, eax
lea rsi, [rbp-0x5]
call 400640 std::char_traits

movsx ecx, BYTEPTR [rbp-0x5]
mov edx, ecx
sub edx, 0x41
mov QWORDPTR [rbp-0x10], rax
mov DWORDPTR [rbp-0x14], ecx
mov DWORDPTR [rbp-0x18], edx
je 400811 <main+0x51>
jmp 4007fd <main+0x3d>
mov eax, DWORDPTR [rbp-0x14]
sub eax, 0x42
mov DWORDPTR [rbp-0x1c], eax
je 400833 <main+0x73>
jmp 400855 <main+0x95>
movabs rdi, 0x601060
movabs rsi, 0x400904

call400630 std::char_traits

mov QWORDPTR [rbp-0x28], rax
jmp 400872 <main+0xb2>
movabs rdi, 0x601060
movabs rsi, 0x400906

call 400630 std::char_traits

mov QWORDPTR [rbp-0x30], rax
jmp 400872 <main+0xb2>
movabs rdi, 0x601060
movabs rsi, 0x400908

call 400630 std::char_traits
mov QWORDPTR [rbp-0x38], rax
xor eax, eax
```

Figure 3.2: Code transformation example: dissassembled version of the code in Listing 3.5 compiled with MSVC, *GCC*, and *clang* compilers

immediate values `imm`, and control registers `ctr`, the assembly operands are encoded numerically according to their types. We consider 10 types of operands during normalization. The complete list of operand types and mnemonic groups used in BinComp is displayed in Table 3.1.

INSTRUCTION MNEMONIC GROUPS		INSTRUCTION OPERAND TYPES	
Feature	**Description**	**Feature**	**Description**
DTR	Data Transfer	reg	General Register
DTO	Data Transfer Address Object	mem	Direct Memory Reference
FLG	Flag Manipulation	bix	Indirect Memory Reference
DTC	Data Transfer Conversion	imm	Immediate Value
ATH	Binary Arithmetic	ifa	Immediate Far Address
LGC	Logical Operation	ina	Immediate Near Address
CTL	Control Transfer	trr	Trace Register
INO	Input/Output	dbr	Debug Register
INT	Interrupt/System	ctr	Control Register
FLT	Floating	otr	Others
MSC	Misc		

Table 3.1: Proposed mnemonic and operand types grouping

3.2.2 Compiler Tags

Compilers may leave behind fingerprints in the form of strings or constants in the produced binaries by default. As an example, binaries compiled with *GCC* compiler contain a tag which survives the symbol stripping process. Likewise, MSVC inserts watermarks of compiler versions in the style of `xor`-encoded values in the file header section. Such values can be considered indicators of compilers and/or versions. Using a parsing mechanism, we extract compiler tag values from `PE`, `ELF`, and `COFF` binaries.

3.2.3 Compiler Functions

The Compiler Functions (`CF`) features are extracted in two steps: i) intersect a set of disassembled binaries, and ii) extract symbolic and numerical feature vectors for each compiler function. We extract the `CF` via the intersection process that will be explained in Section 3.3.2. We employ the code matching and function fingerprinting technique presented in [179] to encode structural, syntactic and semantic features of assembly functions and to facilitate binary comparison. Two feature vectors $\vec{V}_s$ and $\vec{V}_n$ are built for each function in order to capture the symbolic and numerical aspects of the code. We consider all the features presented in Table 3.1 as part of numerical feature vector $\vec{V}_n$ as well as three additional

features, namely CC, BB, KX, that represent cyclomatic complexity [152], number of basic blocks, and number of calls within functions, respectively. The symbolic feature vector V_s is obtained from the debug information in the supervised compilation, which contains function identifiers. These features are listed in Table 3.2.

Symbolic Features		
Constants	Number of Constants	Number of Instructions
Strings	Number of Strings	Number of Operands
Code Refs.	Number of Code Refs.	Number of API Tags
Function calls	Number of Function calls	Number of Library Tags
Imported Functions	Number of Imported Functions	Number of Mnemonic groups
Arguments	Number of Arguments	Size of Arguments
Size of Local Variables	Size of Function	Return Type

Table 3.2: Symbolic features

Example 3.2.2 *An example of the computed $\vec{V}_n$ features for two programs $prog1, prog2$, complied with GCC (O2) compiler under Windows is provided in Table 3.3. The 'symbolic' feature represents the program and function identifiers. The distinct main functions are highlighted in light gray. The table lists the compiler functions along with a subset of feature vectors. As can be seen, functions with similar numerical features are mainly grouped together. As highlighted in dark gray, certain functions with similar symbolic names pertaining to compiler/helper code, exhibit small discrepancies in numerical values due to changes in structural information (e.g., basic blocks and calls). This observation indicates the importance of inexact function signatures (based on similarity calculation of numerical vectors) for compiler function detection and function labelling.*

In order to validate our features selection for compiler fingerprinting, we apply a hierarchical clustering technique based on *Principal Component Analysis* [212] (PCA) to rank the features, as presented in Table 3.3. By sorting the features based on the order of importance, the top-six features are REG, MEM, BB, CTL, CC, and IMM. The obtained results can be interpreted as follows. The CF compiler profile can be attributed based on the types

SYMBOLIC		NUMERICAL																				
P_ID	FUNC_ID	DTR	DTO	FLG	DTC	ATH	LGC	CTL	INO	INT	FLT	MSC	CC	BB	KX	REG	MEM	BIX	BID	IMM	IFA	INA
prog1	_mainCRTStartup	1	0	0	0	1	0	1	0	0	0	0	1	1	2	1	1	1	0	2	0	1
prog1	_WinMainCRTStartup	1	0	0	0	1	0	1	0	0	0	0	1	1	2	1	1	1	0	2	0	1
prog2	_mainCRTStartup	1	0	0	0	1	0	1	0	0	0	0	1	1	2	1	1	1	0	2	0	1
prog2	_WinMainCRTStartup	1	0	0	0	1	0	1	0	0	0	0	1	1	2	1	1	1	0	2	0	1
prog1	___main	1	0	1	0	0	0	3	0	0	0	0	1	4	0	3	2	0	0	1	0	2
prog2	___main	1	0	1	0	0	0	3	0	0	0	0	1	4	0	3	2	0	0	1	0	2
prog1	___mingw_globfree	1	0	0	0	0	0	4	0	0	0	0	1	4	0	1	0	1	1	1	0	2
prog2	___mingw_globfree	1	0	0	0	0	0	4	0	0	0	0	1	4	0	1	0	1	1	1	0	2
prog1	_register_frame_ctor	3	0	0	0	1	0	2	0	0	0	0	1	1	2	4	0	1	0	2	0	2
prog2	_register_frame_ctor	3	0	0	0	1	0	2	0	0	0	0	1	1	2	4	0	1	0	2	0	2
prog1	___do_global_dtors	2	1	1	0	2	0	4	0	0	0	0	3	5	0	15	3	1	2	2	0	2
prog2	___do_global_dtors	2	1	1	0	2	0	4	0	0	0	0	3	5	0	15	3	1	2	2	0	2
prog1	_glob_store_entry.part.2	3	1	1	0	2	1	3	0	0	0	0	1	3	1	40	1	3	10	5	0	2
prog2	_glob_store_entry.part.2	3	1	1	0	2	1	3	0	0	0	0	1	3	1	40	1	3	10	5	0	2
prog1	___chkstk_ms	2	1	0	0	1	1	4	0	0	0	0	3	3	0	11	0	2	1	6	0	2
prog2	___chkstk_ms	2	1	0	0	1	1	4	0	0	0	0	3	3	0	11	0	2	1	6	0	2
prog1	_glob_store_entry	1	0	1	0	0	0	4	0	0	0	0	2	5	0	5	0	0	0	1	0	3
prog2	_glob_store_entry	1	0	1	0	0	0	4	0	0	0	0	2	5	0	5	0	0	0	1	0	3
prog2	_main	3	0	0	0	1	1	2	0	0	0	0	1	1	3	5	0	1	0	3	0	3
prog1	_telldir	1	0	1	0	2	0	4	0	0	0	0	2	4	1	7	0	1	2	4	0	3
prog2	_telldir	1	0	1	0	2	0	4	0	0	0	0	2	4	1	7	0	1	2	4	0	3
prog1	___report_error	2	1	0	0	1	0	1	0	0	0	0	1	1	3	11	1	2	8	4	0	3
prog2	___report_error	2	1	0	0	1	0	1	0	0	0	0	1	1	3	11	1	2	8	4	0	3
prog1	___dyn_tls_dtor@12	1	0	1	0	2	0	4	0	0	0	0	2	4	1	14	0	1	5	8	0	3
prog2	___dyn_tls_dtor@12	1	0	1	0	2	0	4	0	0	0	0	2	4	1	14	0	1	5	8	0	3
prog1	_glob_registry.part.1	3	1	1	0	2	1	5	0	0	0	0	3	4	2	27	1	3	6	5	0	4
prog2	_glob_registry.part.1	3	1	1	0	2	1	5	0	0	0	0	3	4	2	27	1	3	6	5	0	4
prog1	_main	3	1	0	0	2	1	2	0	0	0	0	1	1	4	17	0	3	9	6	0	4
prog2	___gcc_deregister_frame	3	0	1	0	1	0	3	0	0	0	0	3	5	2	15	0	3	1	7	0	4
prog1	___gcc_deregister_frame	3	0	1	0	1	0	3	0	0	0	0	3	5	4	15	0	3	1	7	0	4
prog1	_glob_store_collated_entries	3	0	1	0	2	0	3	0	0	0	0	5	5	4	22	0	2	2	2	0	6
prog2	_glob_store_collated_entries	3	0	1	0	2	0	3	0	0	0	0	5	5	4	22	0	2	2	2	0	6
prog1	_closedir	3	0	1	0	2	0	5	0	0	0	0	3	5	3	19	0	3	2	4	0	6
prog2	_closedir	3	0	1	0	2	0	5	0	0	0	0	3	5	3	19	0	3	2	4	0	6
prog1	_glob_initialise	3	1	1	0	2	1	6	0	0	0	0	5	8	1	34	1	2	6	7	0	6
prog2	_glob_initialise	3	1	1	0	2	1	6	0	0	0	0	5	8	1	34	1	2	6	7	0	6
prog2	____w64_mingwthr_add_key_dtor	3	1	1	0	1	1	5	0	0	0	0	2	5	3	37	3	4	7	8	0	6
prog1	____w64_mingwthr_add_key_dtor	3	1	1	0	1	1	5	0	0	0	0	2	5	5	37	3	4	7	8	0	6
prog1	___do_global_ctors	3	1	1	0	2	1	6	0	0	0	0	5	8	1	18	3	1	2	5	0	7
prog2	___do_global_ctors	3	1	1	0	2	1	6	0	0	0	0	5	8	1	18	3	1	2	5	0	7
prog2	___mingwthr_run_key_dtors.part.0	3	1	1	0	1	0	4	0	0	0	0	5	6	4	30	1	5	3	6	0	8
prog1	___mingwthr_run_key_dtors.part.0	3	1	1	0	1	0	4	0	0	0	0	5	6	8	30	1	5	3	6	0	8
prog1	___dyn_tls_init@12	3	0	1	0	2	2	7	0	0	0	0	6	12	1	32	3	1	5	16	0	8
prog2	___dyn_tls_init@12	3	0	1	0	2	2	7	0	0	0	0	6	12	1	32	3	1	5	16	0	8
prog2	___gcc_register_frame	3	0	1	0	1	0	3	0	0	0	0	6	10	4	29	1	6	3	14	0	9
prog1	___gcc_register_frame	3	0	1	0	1	0	3	0	0	0	0	6	10	8	29	1	6	3	14	0	9
prog2	___write_memory.part.0	3	1	1	0	1	0	5	0	0	0	0	4	9	6	61	0	6	29	12	0	11
prog1	___write_memory.part.0	3	1	1	0	1	0	5	0	0	0	0	4	9	9	61	0	6	29	12	0	11
prog2	__w64_mingwthr_remove_key_dtor	3	0	1	0	1	1	6	0	0	0	0	5	13	4	39	3	6	7	7	0	12
prog1	__w64_mingwthr_remove_key_dtor	3	0	1	0	1	1	6	0	0	0	0	5	13	7	39	3	6	7	7	0	12
prog1	___mingw_glob	3	1	1	1	2	1	6	0	0	0	0	7	10	6	60	0	6	13	13	0	13
prog2	___mingw_glob	3	1	1	1	2	1	6	0	0	0	0	7	10	6	60	0	6	13	13	0	13
prog2	___mingw_TLScallback	3	1	1	0	1	0	7	0	0	0	0	7	13	4	23	6	2	2	12	0	15
prog1	___mingw_TLScallback	3	1	1	0	1	0	7	0	0	0	0	7	13	6	23	6	2	2	12	0	15
prog1	___cpu_features_init	3	0	3	0	0	3	7	0	0	0	0	15	25	0	30	10	0	0	25	0	15
prog2	___cpu_features_init	3	0	3	0	0	3	7	0	0	0	0	15	25	0	30	10	0	0	25	0	15
prog1	_rewinddir	3	1	1	0	2	4	6	0	0	0	0	7	15	4	70	0	6	30	23	0	16
prog2	_rewinddir	3	1	1	0	2	4	6	0	0	0	0	7	15	4	70	0	6	30	23	0	16
prog1	_seekdir	3	1	1	0	2	4	8	0	0	0	0	12	17	4	80	0	6	32	24	0	17
prog2	_seekdir	3	1	1	0	2	4	8	0	0	0	0	12	17	4	80	0	6	32	24	0	17
prog2	_readdir	3	1	1	0	2	4	7	0	0	0	0	11	20	7	91	0	8	34	25	0	22
prog1	_readdir	3	1	1	0	2	4	7	0	0	0	0	11	20	8	91	0	8	34	25	0	22
prog1	__pei386_runtime_relocator	3	1	1	1	2	1	9	0	0	0	0	15	27	6	106	5	6	23	32	0	25
prog2	__pei386_runtime_relocator	3	1	1	1	2	1	9	0	0	0	0	15	27	6	106	5	6	23	32	0	25
prog1	__gnu_exception_handler@4	3	1	1	0	2	2	8	0	0	0	0	18	25	7	42	0	11	10	35	0	28
prog2	__gnu_exception_handler@4	3	1	1	0	2	2	8	0	0	0	0	18	25	7	42	0	11	10	35	0	28
prog1	_opendir	3	1	1	1	3	5	6	0	0	0	0	13	21	11	135	0	19	39	36	0	28
prog2	_opendir	3	1	1	1	3	5	6	0	0	0	0	13	21	11	135	0	19	39	36	0	28
prog1	_glob_strcmp	4	1	1	2	2	4	6	0	0	0	0	32	53	5	154	0	9	30	34	0	50
prog2	_glob_strcmp	4	1	1	2	2	4	6	0	0	0	0	32	53	5	154	0	9	30	34	0	50
prog1	_glob_in_set	3	1	1	2	2	2	7	0	0	0	0	42	60	0	147	0	12	13	44	0	51
prog2	_glob_in_set	3	1	1	2	2	2	7	0	0	0	0	42	60	0	147	0	12	13	44	0	51
prog2	___mingw_CRTStartup	3	1	1	2	3	6	7	0	0	0	0	42	63	20	216	15	22	69	68	0	71
prog1	___mingw_CRTStartup	3	1	1	2	3	6	7	0	0	0	0	42	63	24	216	15	22	69	68	0	71
prog1	_dirname	3	1	1	1	2	3	10	0	0	0	0	44	68	17	215	5	31	90	57	0	73
prog2	_dirname	3	1	1	1	2	3	10	0	0	0	0	44	68	17	215	5	31	90	57	0	73
prog1	_glob_match	6	1	1	2	2	3	8	0	0	0	0	66	99	32	389	0	43	140	75	0	113
prog2	_glob_match	6	1	1	2	2	3	8	0	0	0	0	66	99	32	389	0	43	140	75	0	113

Light gray: Main functions in different programs. *Dark gray*: The same compiler functions in distinct programs may have slightly different numerical feature vectors.

Table 3.3: Numerical and symbolic features

of registers selected by the compiler and the method of memory access, followed by the structural complexity (basic blocks) which point to the compiler family's behaviour.

3.2.4 Annotated Control Flow Graph

A control flow graph (`CFG`) mainly accounts for the structure of functions, however, dissimilar functions may have a similar structure. This can result in high false positive rates if it is used as the only feature for function detection. Therefore, we propose an additional graph-based representation, namely the Annotated Control Flow Graph (`ACFG`), as an extension of `CFG`. The purpose of this graph is to facilitate detection of compiler versions and optimization levels based on function compositions. We transform each disassembled compiler/linker function into a graph and build the `ACFG` profiles based on the mnemonics, operands and function calls in the instructions per basic block. The control flow semantics as well as the encoded instructions are captured by `ACFG`, which carries the required information for precise compiler function identification. Consequently, semantically similar compiler functions will be translated into equivalent `ACFG` profiles. More formally,

Definition 1 *An Annotated Control Flow Graph, $ACFG$, is defined as an attributed graph (V, E, ζ, γ), where V denotes a set of vertices representing function basic blocks that contain the instructions, $E \subseteq (V \times V)$ is a set of edges representing the jump instructions, γ is an instruction clustering function, and ζ is a node coloring function based on operand types. Functions γ and ζ are used during the CFG normalization and the encoding of control flow graphs and subgraphs.*

The process of `ACFG` generation involves the following steps. For each function `CFG`, we compute the operation code frequencies per assembly instruction groups. As shown in Table 3.1, each instruction can be grouped based on mnemonic semantics and operand types. We reduce the features according to Table 3.4 and categorize x86 instructions into

six groups for graph encoding. We then complement the ACFG with encoded subgraph values and include it in the structural profiles of compilers.

Feature Group	Description	Example
DTR & STK	Data Transfer and Stack	push, mov, xchg
ATH & LGC	Arithmetic and Logical	add, xor
CAL & TST	Call and Test	call, cmp
REG & MEM	Register and Memory	esi, [esi+4]
REG & CONST	Register and Constant	esi, 30
MEM & CONST	Memory and Constant	[esi+8], 20

Table 3.4: Instruction patterns for annotation

Example 3.2.3 *We illustrate the CFG and ACFG representations of a randomly selected GCC compiler function, i.e., __do_global_ctors in Figure 3.3. Each basic block is transformed and normalized based on a set of features describing the type, category, and frequency of instructions (mnemonics and operands).*

The details of the ACFG construction are presented in Algorithm 1. First, we normalize the assembly instructions of a given CFG. Each assembly instruction can be categorized according to the groups of mnemonics, types of operands (e.g., immediate value, register, memory reference, etc.), and types of function calls, amongst others. The helper function GetInstructionGroup() in line 17 returns the groups of instructions for individual opcodes. The NormalizeOperandType() function in line 31 returns the associated type of the operands. Similarly, the destination of function calls could also be categorized according to their types. Call destinations can be internal or external to the disassembly; System and API calls are also grouped into multiple classes based on their side effects on the target systems. The GetFunctionCategory() function in line 34 returns the general context of functions and includes categories, such as file, network, registry, crypto, service, and memory. Each category contains OS-specific API functions.

Algorithm 1: CFG normalization for ACFG generation

```
1  !t Input: G = (BB, E) : CFG of an assembly function.
   Output: G' = (BB', E'): Norm-CFG, Lists A_CNT, G_CNT, C_CNT, and counts T, B.
2  Initialization;
3  NCFG ← init(graph);
4  A_CNT ← init(key, value) ;                          // Instruction count
5  G_CNT ← init(key, value) ;                          // Group count
6  C_CNT ← init(key, value) ;                          // Call count
7  T ← 0 ;                                             // Number of instructions
8  B ← 0 ;                                             // Number of basic blocks
9  begin
10     Perform breadth-first traversal (BFT) of the CFG basic blocks;
11     foreach unvisited basic block bb ⊆ BB do
12         Create a node and add the corresponding basic block bb' to Norm − CFG;
13         B ← B + 1;
14         foreach instruction INST ⊆ bb do
15             T ← T + 1;
16             Add a new instruction placeholder P to bb';
17             foreach mnemonic M ∈ INST do
18                 G_M ← GetInstructionGroup(M) ;      // Get the group label
19                 Place G_M in P as the group of mnemonic;
20                 if M ∉ A_CNT list then
21                     Add M to the A_CNT list as key;
22                     A_CNT[M] ← 1;
23                 else
24                     A_CNT[M] ← A_CNT[M] + 1;
25                 if G ∉ G_CNT list then
26                     Add G to the G_CNT list as key;
27                     G_CNT[G] ← 1;
28                 else
29                     G_CNT[G] ← G_CNT[G] + 1;
30             foreach operand O ∈ INST do
31                 if mnemonic (M ∈ INST) ≠ call then
32                     TP ← NormalizeOperandType(O); // Get the operand type
33                     Place TP in P as the operand of G;
34                 else
35                     FC ← GetFunctionCategory(O) // Get the category
36                     Place FC in P as the operand of G;
37                     if FC ∉ C_CNT list then
38                         Add FC to the C_CNT list as key; // Set the count value
39                         C_CNT[FC] ← 1;
40                     else
41                         C_CNT[FC] ← C_CNT[FC] + 1; // Increment the count value
42         Flag the basic block bb as visited;
43         foreach in/out edge (e_i^bb, e_o^bb) ⊂ E in CFG do
44             Create the corresponding in/out edge (e_i^bb', e_o^bb') to bb' in NormCFG;
45         Continue traversal to the next basic block bb;
46     return Norm − CFG, A_CNT, G_CNT, C_CNT, T, B;
```

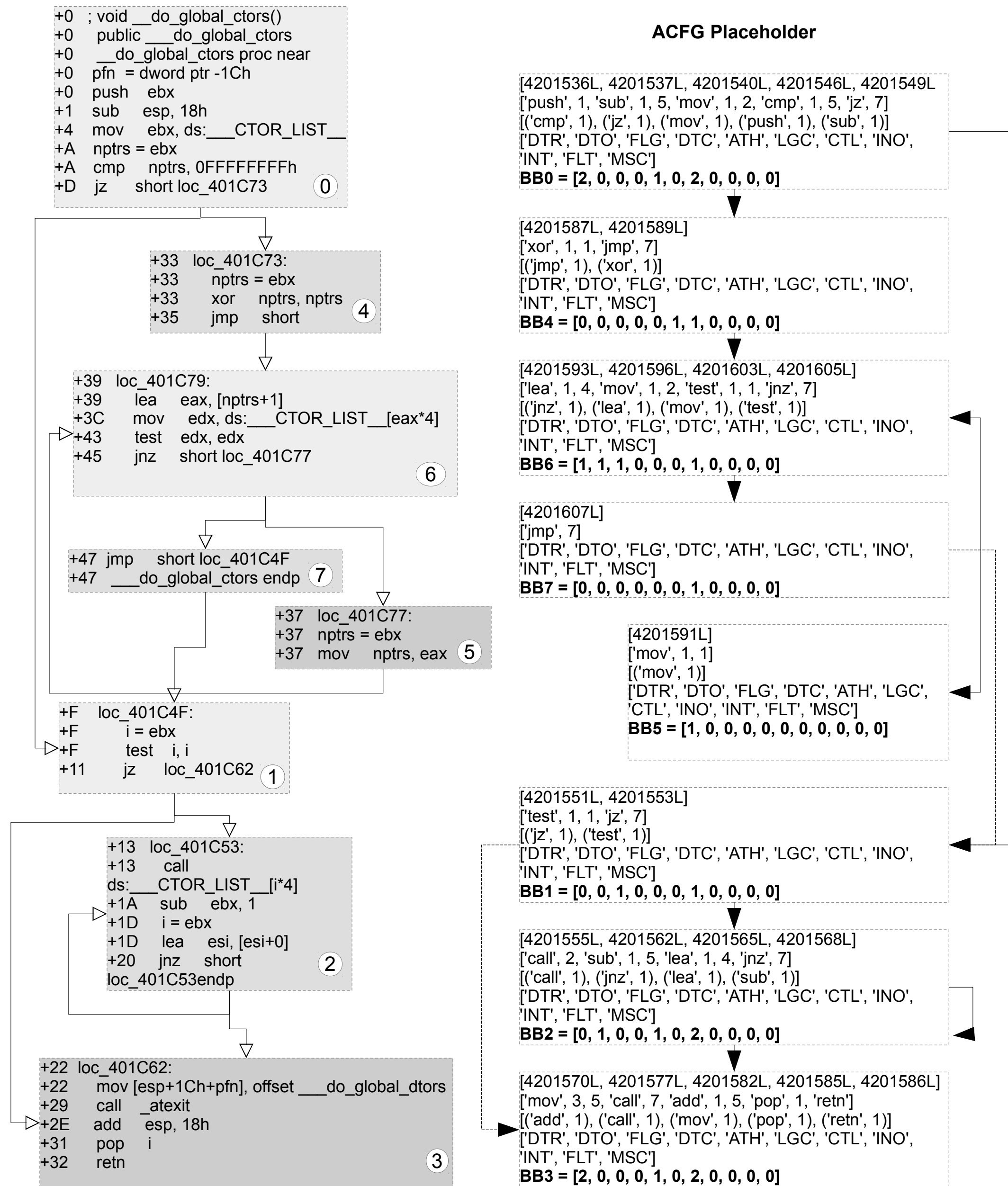


Figure 3.3: ACFG construction of `__do_global_ctors` compiler function

3.2.5 Compiler Constructor and Terminator

The compiler-specific call sequences can be identified by analyzing program call graphs and control flow graphs. Each compiler family behaves differently in setting up the program startup code, initialization routines, and program termination (OS transition) code.

The program initialization and termination processes entail multiple compiler functions that manage the preparation/cleanup of the call stacks, required libraries, and OS interactions before and after the execution of main function. Multiple function calls can be observed during these processes. As part of the compiler constructor and terminator (`CCT`) profile, we capture features such as the sequence of initialization/termination code, number of function basic blocks, and properties of caller/callee functions on the call graph.

In order to generate `CCT` signatures, we traverse the program call graph until reaching the main function. We store the sequence of function calls and function fingerprints of the pre/post main (initialization and termination) and call patterns to imported libraries as part of the compiler profile. Figure 3.4 provides an example of a constructor and terminator graph for the *GCC* compiler.

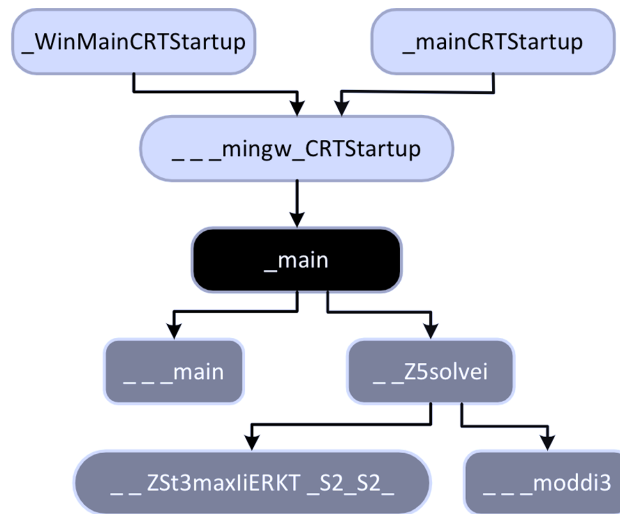


Figure 3.4: Sample of *GCC* compiler constructor and terminator

3.3 Multi-layered Compiler Provenance Attribution

This section presents the proposed multi-layered approach to compiler fingerprinting, identification and behavorial profiling. The methodology is broken down into three layers,

namely *compiler family identification*, *compiler function fingerprinting and labelling*, and *compiler version and optimization detection* as depicted in Figure 3.1. Each layer consists of one or more processes that support the compiler detection heuristics.

3.3.1 Layer 1: Compiler Family Identification

Building Compiler Transformation Profile (`CTP`) and assigning Compiler Tags (`CT`) followed by the detection of the compiler family are the main objectives of Layer 1. A supervised compilation process is applied to a collection of known source code, and output binaries are analyzed to populate the `CTP` and `CT` profiles. These profiles are subsequently processed using pattern matching techniques to identify compiler families.

The proposed behavioral profiles conform with standard compilation steps of C-based family of compilers as shown in Figure 3.5. In this figure, the labelled states (source F_S, binary F_B, exported library F_E) indicate feature extraction points and the blocks represent specific stages of the process. First, the input source code containing the code and data patterns are defined. The supervised compilation process yields compiler-specific transformation profiles that enable assembly to source code matching. This process is repeated using various combinations of chosen source components. This may include crafted excerpts of code and data patterns for control flow, branching, loops statements, memory accesses, calling conventions, system calls, and composite/abstract data types.

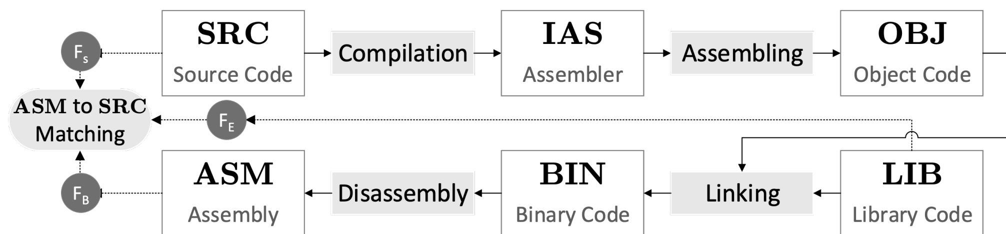


Figure 3.5: Supervised compilation steps and produced artifacts

We extend an existing assembly to source code matching approach, namely, BINSOURCERER[5] [179], by combining syntactic features with structural and semantics features to build CTP profiles as follows:

1. **Initialization**: The input of this step is the original source code (SRC), and the output is a pre-processed source file that includes the contents of header files imported into the source code. The symbolic constants are replaced with their values. To support the signature generation process, the values of constants and the list of source-level functions along with their prototypes are extracted as part of the source features F_S.

2. **Compilation**: The compilation step takes the target platform specifications and the expanded source file generated in previous step as inputs. Then it generates a platform-dependent assembly file (IAS) according to the compilation settings, such as code generation parameters and optimization level. The list of assembly functions is then extracted in support of the signature generation process.

3. **Assembling**: The input assembly file (IAS) is translated into machine object code (OBJ), which in turn gets disassembled for feature extraction F_E. In support of signature generation, each assembly-level function is matched against the respective source-level function extracted during the first step. Subsequently, function-level transformation (mapping) profiles are created between assembly code patterns and source language statements. These profiles provide indications on syntactic styles and transformation rules of the source compiler.

4. **Linking**: The output binary file (BIN) is built by linking the library object code (LIB) with the object code (OBJ) generated in the preceding step. The produced file may include the code (statistically linked) or references (dynamically imported) of the library functions called in the source program.

[5] https://github.com/BinSigma/BinSourcerer. Accessed on Dec 20, 2020.

5. **Disassembling**: The binary file gets disassembled into assembly (ASM) file listings. In addition to the assembly-level functions extracted from the object file (OBJ) in Step 3, the disassembly at this stage includes extra helper/utility functions inserted by the linker, which are used for signature generation (F_B). Furthermore, the disassembly is utilized to extract compiler tags.

The code transformation profiles describe compiler behavior in translating code/data patterns to binary representations. These profiles are matched against target assembly functions in order to identify the most likely compiler, based on binary features F_B.

Detection Method

Suppose program P disassembly is composed of functions $F = \{f_1, \ldots, f_n\}$. The file-level `CT` features are represented as a list of strings. The function-level `CTP` features for each $f_i \in F$ are represented with a feature vector $\vec{v}_{f_i} \in \vec{V}$ that captures the available `CTP` features of f_i in the form of key/value pairs i.e., (k_j, val_j): $\vec{v}_{f_i} := \langle (k_1, val_1), \ldots, (k_n, val_n) \rangle$. These features capture the compiler transformation patterns based on predefined (chosen) data/code components (e.g., loops, branching, and arrays). For each compiler c_i, we generate a signature Sig_{c_i}, which represents the behavior profile. A combination of exact and threshold-based matching techniques is used to compare the extracted `CTP` and `CT` features of target P program against the Sig_{c_i} profiles, resulting in the identification of the most similar compiler family c_i.

3.3.2 Layer 2: Compiler Function Labelling

The purpose of Layer 2 is to identify and label compiler-related functions. Therefore, the Compiler Function (`CF`) features are utilized. The proposed approach, shown in Figure 3.6, is based on pairwise comparison and intersection of program disassemblies to identify function groups. More specifically, we leverage the supervised compilation process (as

explained in Section 3.3.1) to compile a set of programs using compiler c_i. Following the standard compilation, assembling, linking, and disassembly processes, we obtain a pair of disassembled program functions $\langle F_{P_1}, F_{P_2}\rangle$, as well as their respective symbolic ($\vec{V_s}$) and numerical ($\vec{V_n}$) feature vectors (CF feature).

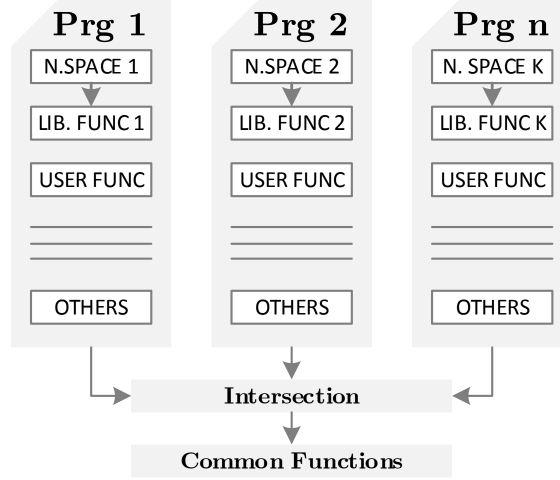


Figure 3.6: Intersecting programs to obtain common compiler functions

In the general case, we define the whole set of disassembly functions of program P_i as the union of five types of assembly functions i.e., $F_{P_i} := (F_{S_i} \cup F_{D_i} \cup F_{E_i} \cup F_{H_i} \cup F_{U_i})$, namely, the set of statically linked functions F_{S_i}, dynamically imported functions F_{D_i}, exported functions F_{E_i}, helper/utility functions F_{H_i}, and user functions F_{U_i}. An effective function fingerprinting technique would be required to partition the set of F_{P_i} into five disjoint subsets, each representing a unique type of assembly functions. In this chapter, we focus to identify the helper/utility functions F_{H_i}.

After intersecting the list of assembly functions $\langle F_{P_1}, F_{P_2}\rangle$ and computing the CF-related feature vectors, we use the numerical component of feature vectors ($\vec{V_n}$ and $\vec{V_s}$) to measure the pairwise similarity of assembly functions $Sim(f_i, f_j), \forall_i \in F_{P_1}, \forall_j \in F_{P_2}$ based on vector distances. To this end, the k-means algorithm [82] is utilized to cluster similar functions together. The functions with similar attributes $(v_{n_1}, \ldots, v_{n_z})$ are clustered into k clusters $C = C_1, \ldots, C_k$ where $(k \leq |F_{P_1}| + |F_{P_2}|)$ minimizing the intra-cluster sum

of squares. The mean of the numerical feature values of each cluster is denoted by μ_i in the following equation:

$$\arg\min_{C} \sum_{i=1}^{k} \sum_{v_j \in C_i} \|v_j - \mu_i\|^2 \quad (1)$$

Afterwards, the symbolic feature vector ($\vec{V_s}$) is used for cluster validation when function identifiers are available (e.g., debug binaries in supervised compilation). In such cases, the common identifiers can be used as the basis for function list intersections, i.e., we compute $list\Big(set\big(\vec{V}_{P_1}.names()\big) \cap set\big(\vec{V}_{P_2}.names()\big)\Big)$, where the $list()$ and the $names()$ functions return the list of disassembled functions and the function name identifiers, respectively.

Given a target program P_t with an unknown compiler, we pass it through the disassembler and get a set of unknown functions $F_{P_t} = \{f_1, \ldots, f_m\}$. Then, we leverage the `CF` function identification technique to determine the types of assembly functions $f_i \in F_{P_t}$, from the set of $F_{P_t} := (F_{S_t} \cup F_{D_t} \cup F_{E_t} \cup F_{H_t} \cup F_{U_t})$, based on the similarity calculations on $V_s = \{V_{s_1}, V_{s_2}, \ldots, V_{s_z}\}$ and $V_n = \{V_{n_1}, V_{n_2}, \ldots, V_{n_z}\}$. The vectors are compared against reference clusters of compiler profiles and the most similar functions are grouped together. Once the clusters are formed, the labels of the most similar compiler/linker function will be assigned to each cluster centroid.

Detection method

In this section, we present a function fingerprinting approach for signature generation and detection from assembly functions. We define a feature extraction function as $X : F \to V$, where F is the set of helper/utility functions and V is the set of all possible features, i.e., $v_i = X(f_i)$. The input to this function is a program P comprised of a set of assembly functions $F = \{f_1, \ldots, f_m\}$, and the output is a set of features $v_i \in V$. The output feature vectors can be partitioned into two subsets of V_s and V_n.

Function G defined as $G : V \to S$ takes the set of features to generate the fingerprints.

The encoded characteristics of function $f_i \in F$ is represented by each fingerprint $s_i \in T$:

$$s_i = G(v_i) = G(X(f_i)) \tag{2}$$

This function behaves similar to a hash function, compressing the variable-length input assembly vector into a fixed length signature digest. However, the output is based on a normalized version of assembly opcodes and types of operands. We interpret this function as a semantic hash that creates a bit vector of length n from the subset of features V from f_i. The hashing technique and its sensitivity can be defined based on the feature space size and number of input bits.

$$s_i \in domain(V) \rightarrow \{0,1\}^n \tag{3}$$

Function fingerprinting depends on pairwise similarity comparisons for assembly functions. It also considers incremental clustering in which the similarity of a target function is calculated against a reference group. Function M assigns a similarity score to a pair of candidate vectors, either at the level of fingerprints, i.e., $M : S \times S \rightarrow \mathbb{R}^+$, or at the level of function features $M : V \times V \rightarrow \mathbb{R}^+$.

The Jaccard similarity is utilized for quantifying and measuring the distance between a candidate fingerprint pair (s_i, s_j) for functions (f_i, f_j), computed as follows [93]:

$$dist_J(s_i, s_j) = \frac{B(s_i \wedge s_j)}{B(s_i \vee s_j)} \tag{4}$$

where function B returns the number of set bits in the fingerprint vector.

The functionality of this layer can be abstracted using a labelling function C that takes the set of function fingerprints (for target T_t) and the reference S_r fingerprints, and returns a label $l_i \in L$ based on the corresponding compiler, i.e., $C : S_t \times S_r \times N \rightarrow L$. In this relation, N is a mapping matrix that is learned during the supervised compilation process. This matrix links compilers to specific helper/utility function profiles based on similarity

analysis. The labels obtained from this layer identify compiler-specific functions in target disassemblied binaries.

3.3.3 Layer 3: Version and Optimization Recognition

The objective of Layer 3 is to detect more refined compiler properties, such as optimization level and version. Similar to previous layers, we leverage the supervised compilation process to generate features. However, this layer deals with graph-based structural and semantic features of Annotated Control Flow Graph (`ACFG`) and Compiler Constructor and Terminator (`CCT`) profiles. Empirically, these two types of features are the most representative of optimization heuristics and specific differences in compiler versions. We use neighborhood hashing [93], as well as graph encoding to adjust the granularity levels of binary components in support of fingerprint generation.

Detection method

The subgraph matching technique is helpful in matching `CCT` profiles as it allows us to compare the initialization and termination flows with the reference patterns of compiler behavior. A common method for graph comparison is based on canonical encoding of graph structures. For the purpose of this layer, we combine subgraph encoding with neighborhood hashing techniques to reach the required granularity levels.

A neighborhood hash graph kernel (NHGK) [93] is applied to subsets of the call graph or an `ACFG` to generate fingerprints of adjacent nodes. In this scheme, function G maps an input annotated graph AG to a bit vector of length m:

$$G : AG \to \{0, 1\}^m \tag{5}$$

The neighborhood hash value h for a target function f_t and the corresponding set of

immediate neighbor functions N_{f_t} can be obtained as follows [104]:

$$h(f_t) = shr_1(G(f_t)) \oplus (\oplus_{f_k \in N_{f_t}} G(f_k)) \quad (6)$$

where shr_1 indicates a single-bit shift to the right and $\oplus$ denotes the XOR function.

3.4 State-of-the-Art Compiler Provenance Extraction

In this section, we provide an overview of the approach proposed in [187] that will be referred to as ECP (which stands for Extracting Compiler Provenance) in this chapter. We re-implement the proposed approach and then present our obtained evaluation results, discuss the details of our findings, and highlight the limitations of ECP.

3.4.1 Overview

The ECP approach models compiler identification as a structured classification problem and labels each byte of the binaries with the information on whether it is compiled with one or two compilers (statically linked code). The authors utilize *wildcards idiom* features, which are defined as short sequences of instructions that neglect details, such as literal arguments and memory offsets. After extracting the idiom features from a large number of binaries, the authors consider top-ranked features based on the results of mutual information (MI) computation between the idiom features and compiler classes. The results of the MI indicate how frequent the code originated from a particular compiler contains a specific idiom. A linear-chain Conditional Random Field [153] model is trained to assign high probabilities to correct compiler classes.

3.4.2 Dataset Generation

The proposed technique is performed on three sets of binaries containing code from three compilers: GNU C Compiler (*GCC*), Intel C Compiler (*icc*), and Visual Studio (*MSVC*). The dataset is collected for *GCC* and *MSVC* on Linux and Windows workstations, respectively. In addition, various open-source software packages are compiled with the *icc* compiler. Most of the aforementioned binaries are not publicly available. Therefore, we collect different files from different sources to build our dataset. We generate the first dataset (called G dataset) by collecting the programs related to different years (2008-2014) of the *Google Code Jam*[6] competition. The second dataset, called U, is composed of several source code samples from our university. We compile all the source code with three compilers that are used in ECP, namely, *GCC*, *icc*, and *MSVC*, as well as one additional compiler, namely *clang* (Xcode). We build our dataset after considering all possible combinations of compilers, and optimization levels on various combinations of the binaries.

3.4.3 Evaluation Results

In this section, we examine both accuracy and efficiency of ECP by performing different experiments. To this end, we split the training data into ten sets, reserving one set as a testing set. Then, we train the classifier on the remaining sets, and evaluate its accuracy on the testing set. More specifically, we perform different experiments based on the following scenarios: (i) changing the number of files and the type of dataset; (ii) modifying the threshold value of the top-ranked features; (iii) mixing various percentages of binaries taken from different datasets in order to observe the effect of diverse binaries; and (iv) measuring the time with respect to the number of features. In what follows, we discuss our results based on the aforementioned scenarios.

Accuracy. The ECP approach can attain relatively high accuracy, as shown in Figure 3.7,

[6] https://codingcompetitions.withgoogle.com/codejam/archive. Accessed on Dec 20, 2020.

where the accuracy for the G dataset is between 89% and 98%. This is higher than the accuracy obtained using the U dataset (min = 82%, max = 95%). By analyzing the source code, we find out that the user contribution in the U data set is greater than that of the G dataset. The U dataset is also more complex than the G dataset, as it consists of more advanced code structures, classes, methods, etc.

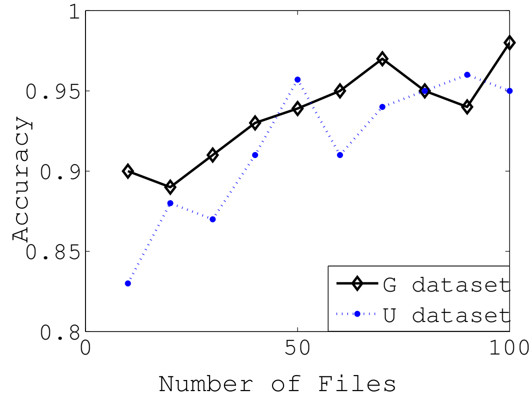


Figure 3.7: ECP accuracy results

Impact of Threshold Value. We study the effects of threshold value on the accuracy by changing the number of top-ranked features. We observe that the accuracy depends on the choice of threshold value, as shown in Figure 3.8. For instance, the best threshold values to correctly identify the *GCC* and *clang* compilers are $16,000$ and $18,000$, respectively.

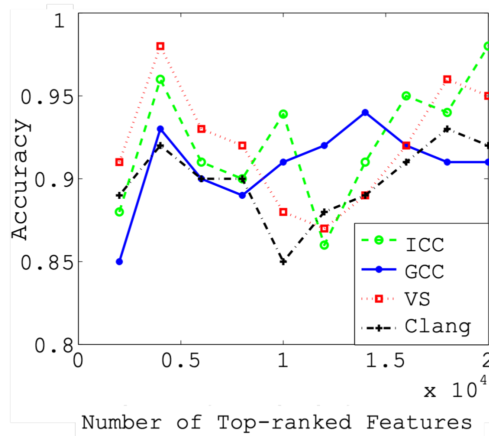


Figure 3.8: Impact of the threshold value

Impact of Dataset. We further conduct experiments to study the effects of diverse datasets on the accuracy. We consider different percentages of mixed datasets and measure the accuracy; we start from 100% G dataset, then change it to 20% G - 80% U datasets, and so forth. As illustrated in Figure 3.9, the accuracy decreases when the diversity of the dataset increases, especially when the percentage of university projects increases, since the user contribution in the U dataset is higher than that of the G dataset.

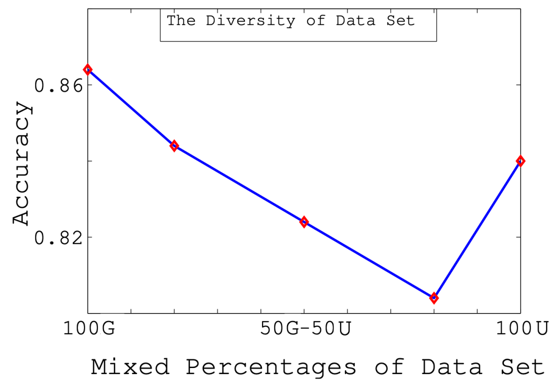


Figure 3.9: Diversity in datasets

Efficiency. We measure the time efficiency of the ECP approach by calculating the total required time for feature extraction and feature ranking. Figure 3.10 shows the total time versus the number of employed features for each experiment.

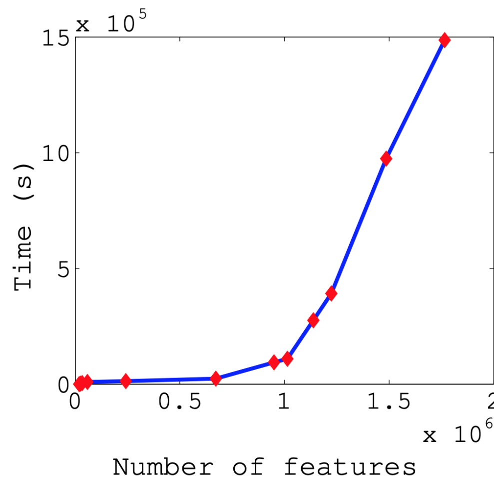


Figure 3.10: Feature extraction and feature ranking time

3.4.4 Discussion

The ECP approach represents a pioneering effort on compiler identification and may attain relatively high accuracy. This approach can also identify the compilers of binaries containing a mixture of code from multiple compilers, such as statically linked library code. However, a few limitations can be observed. First, as the number of features increases, the running time may increase rapidly. Second, the features of specific compilers may only become apparent after examining a large number of binaries. Third, the accuracy may depend on both the dataset and the choice of the threshold.

3.5 Evaluation

In this section, first we introduce our dataset and then provide the evaluation results of BinComp. Finally, we provide the performed comparison results.

3.5.1 Dataset Preparation

Gathering a data corpus for the evaluation of compiler provenance attribution is challenging. For example, despite the fact that collecting code from open-source projects may not be challenging, the source files usually have numerous dependencies which complicates the compilation process. Nonetheless, we choose four free open-source projects to test BinComp. In addition, we gather the programs written for the *Google Code Jam*[7] competition as well as course projects from a programming course at our university. We generate the binaries to build our dataset by compiling the source code with possible combinations of compiler versions and optimization levels (`O0` and `O2`) as shown in Table 3.5. Our dataset consists of $1,177$ binaries, 232 of which belong to *Google Code Jam* dataset, 933 of which belong to *Students Course Projects*, and 12 of which belong to Open-source Projects.

[7] `https://codingcompetitions.withgoogle.com/codejam/archive`. Accessed on Dec 20, 2020.

Compiler	Version	Optimization	Version	Optimization
GCC	3.4	`O0`	4.4	`O2`
ICC	10	`O0`	11	`O2`
MSVC	2010	`O0`	2012	`O2`
Clang	5.1	`O0`	6.1	`O2`

Table 3.5: Different compilers and compilation settings used to build the dataset

3.5.2 Accuracy Results

We evaluate the proposed compiler provenance approach using the aforementioned datasets. We split the training data into ten sets, reserving one set as a testing set, and using nine sets as training sets to evaluate our approach; we repeat this process 1000 times. Since the application domain is characterized by a heightened sensitivity to false positives than false negatives, the F-measure is employed to evaluate the proposed approach as follows:

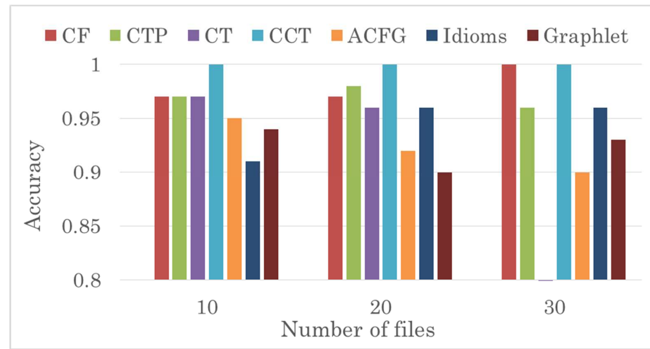
$$F_{0.5} = 1.25 \ . \ \frac{P.R}{0.25P + R}, \ Precision(P) = \frac{TP}{TP + FP}, \ Recall(R) = \frac{TP}{TP + FN} \quad (7)$$

The obtained results of $F_{0.5}$ measure are summarized in Table 3.6.

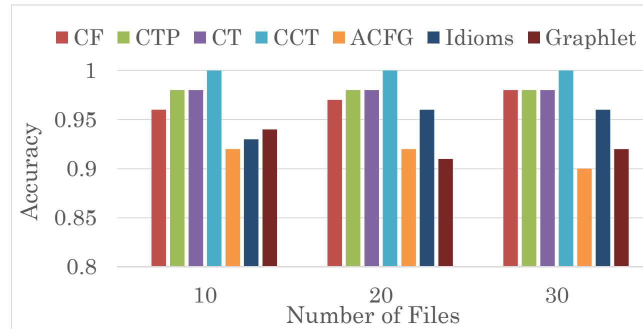
Feature	$F_{0.5}$ **(500 files)**	$F_{0.5}$ **(1000 files)**
Idioms	0.789	0.812
Compiler Transformation Profile (`CTP`)	0.694	0.708
Compiler Constructor & Terminator (`CCT`)	0.807	0.877
Compiler Tags (`CT`)	0.689	0.700
Annotated Control Flow Graph (`ACFG`)	0.634	0.671

Table 3.6: F-measure results

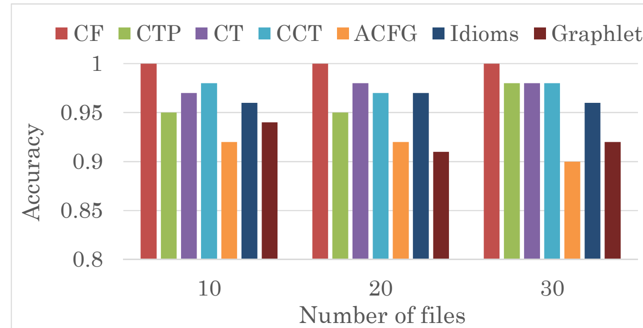
Impacts of different compilers. We test BinComp against different compilers as shown in Figure 3.11. As depicted in Figure 3.11a, BinComp can detect *MSVC* compiler with an average accuracy of 97%, while ECP average accuracy is 93%. The main explanation for this difference lies in the type of features; BinComp uses different kinds of features (syntactical, structural, and semantic), whereas ECP uses only idioms features.



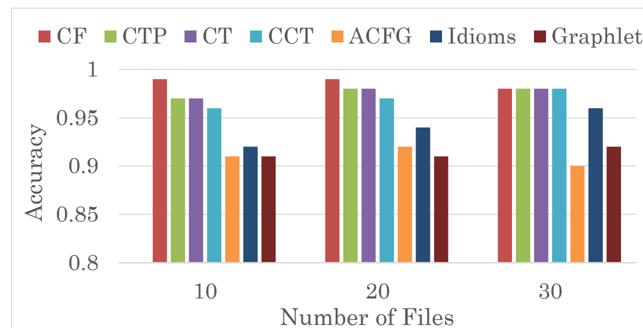
(a) MSVC



(b) ICC



(c) GCC



(d) CLang

Figure 3.11: The accuracy results against different compilers

Impacts of different compiler versions and optimization settings. We further test BINCOMP using different compiler versions and optimization levels, as shown in Table 3.7 and Table 3.8. Table 3.7 indicates that identifying the version of compiler is significantly more difficult than recognizing the compiler and optimization levels. For instance, the accuracy to identify the version of *clang* compiler is below 80%. In addition, we observe that the features of *MSVC* and *clang* compilers are slightly different when we change either the versions or the optimization levels, which makes the detection process more challenging. However, the *GCC* and *icc* compilers produce more diverse code amongst compiler versions compared to the *MSVC* and *clang* compilers.

Compiler	Version	Accuracy	Version	Accuracy
GCC	3.4.x	86%	4.4.x	89%
ICC	10	83%	11.x	90%
MSVC	2010	70%	2012	71%
Clang	5.x	78%	6.1	74%

Table 3.7: Accuracy results for variations of compiler versions

We find that up to 75% of the functions in our dataset are identical when generated by *MSVC* version 2010 or 2012 and with the same optimization level. In other words, the code generator in Visual Studio has remained relatively stable between these versions, which offers an explanation for the low accuracy for *MSVC* version detection. Similarly, we find that up to 85% of the functions of *clang* compiler are identical between two versions. However, we observe changes in our proposed features for the *GCC* and *icc* compilers, which allow to detect the version and optimization level more accurately.

Compiler	Optimization Level	Average Accuracy
GCC	`O0,O2`	91%
ICC	`O0,O2`	89%
MSVC	`O0,O2`	95%

Table 3.8: Accuracy for variations of compiler optimization levels

3.5.3 Comparison

We perform a comparative study amongst BINCOMP, ECP, and IDA PRO on four open-source libraries: `SQLite`, `libpng`, `zlib`, and `OpenSSL`. The obtained results are illustrated in Figure 3.12. We compile `SQLite` and `zlib` with *MCVS* 2010 (`O2`), and the compiler is successfully identified by all three approaches; whereas `libpng` and `OpenSSL` are compiled with *GCC* (`O2`), for whom IDA PRO is not able to identify the source compiler. On the other hand, BINCOMP and ECP provide similar accurate results for all cases, however, ECP dose not identify compiler versions, optimizations and compiler-related functions. Moreover, it requires large dataset and is less efficient than BINCOMP.

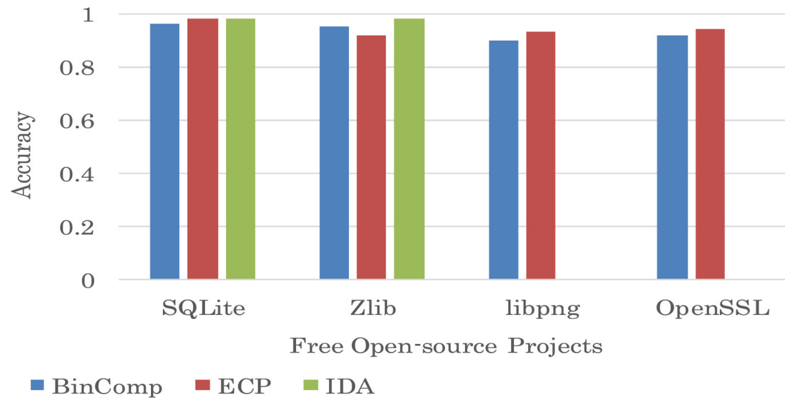


Figure 3.12: Accuracy results comparison amongst BINCOMP, ECP, and IDA PRO

3.6 Limitations and Concluding Remarks

In this chapter, we presented BINCOMP, an automated and accurate approach to recover compiler provenance of program binaries. It extracts syntactic, semantic, and structural features from program binaries for building representative and meaningful signatures to capture compiler characteristics. For instance, the binaries that are compiled with the same compiler have the same compiler tag (`CT`) feature. Furthermore, we formulated the compiler transformation profile (`CTP`) that maps the instructions underlying a binary according

to the most likely compiler family by which it was probably generated. Moreover, we introduced features that capture detailed provenance at function-level granularity, allowing us to recover the optimization levels. We designed annotated control flow graph (ACFG) and compiler constructor and terminator (CCT) features to explicitly capture such changes and identify the version. We evaluated BINCOMP for compiler provenance recovery on different datasets across several compiler families, versions, and optimization levels. Our results demonstrate that compiler provenance can be extracted accurately and efficiently, and thus the proposed approach can be considered as a practical solution for real-world binary analysis.

However, BINCOMP suffers from some limitations. (i) Solely the Intel x86/x86-64 CPU architecture and C++ program language are considered in this work. (ii) We assume that the binary code is not stripped, even though BINCOMP supports limited statistical analysis on stripped binaries via numerical vectors. (iii) Similar to most existing methods, BINCOMP works under the assumption that the binary code is already unpacked and de-obfuscated. One possible future work in this area can be particularly based on the machine learning-based approaches. This could benefit from the set of high-value features introduced in BINCOMP to build more advanced detection models.

Chapter 4

Library Function Identification

Program binaries typically contain a significant amount of library functions taken from standard libraries or free open-source software packages. The capability of automatically identifying such library functions not only enhances the quality and efficiency of threat analysis and reverse engineering tasks, but also improves their accuracy by avoiding false correlations between irrelevant code bases. Furthermore, such automation has a positive impact on other applications, such as clone detection, function fingerprinting, authorship attribution, vulnerability analysis, and malware analysis. This chapter presents BINSHAPE, a scalable approach to identify standard library functions in program binaries, which is based on a robust signature derived from heterogeneous features called *function shape*.

The chapter is organized as follows. First, the library function identification problem, and the approach overview are discussed in Section 4.1. Then, the feature extraction process and the detection methodology are presented in Sections 4.2 and 4.3, respectively. Next, the evaluation of the proposed technique is presented in Section 4.4. Finally, the limitations of BINSHAPE along with the concluding remarks and related future research directions are presented in Section 4.5.

4.1 Introduction

With the rapid development of the information technology, modern software contain a significant amount of third party library functions taken from standard library functions or free open-source software packages. Automatically identifying such library functions enhances the efficiency and accuracy of the binary analysis task.

Automating the process of accurately identifying library functions in binary programs poses the following challenges: *(i) Robustness:* the distortion of features in the binary file may be attributed to different sources arising from the platform, compiler, or programming language, each of which may change the structures, syntax, or sequences of features. Hence, it is challenging to extract robust features that would be less affected by different compilers, slight changes in the source code as well as obfuscation techniques. *(ii) Efficiency:* another challenge is to efficiently extract, index, and match features from program binaries in order to detect a given target function within a reasonable time, considering the fact that many known matching approaches have a high complexity [144]. *(iii) Scalability:* due to the dramatic growth of software packages as well as malware binaries, threat analysts and reverse engineers deal with large numbers of binaries on a daily basis. Therefore, designing a system that could scale up to millions of binary functions is an absolute necessity. Accordingly, it is important to design efficient data structures to store and match against a large number of candidate functions in a repository.

To address the library identification problem, security researchers elaborated techniques to automatically identify library functions in binaries. For instance, the widely-used IDA FLIRT [73] applies signature matching to patterns generated according to the first invariant byte sequence of the function. This simple method is indeed very efficient but the robustness of such solutions still falls short. For instance, IDA FLIRT suffers from the limitation of signature collisions, since in modern C++ libraries there exists functions that

can be absolutely identical at the first bytes but being different. Moreover, various compilers and build options usually affect byte-level patterns, and therefore a new signature for each new version as the result of a slight modification might be required. Similarly, most other existing methods, e.g., UNSTRIP [114] (which is based on the interaction of wrapper functions with the system call interfaces) and LIBV [177] (that employs data flow analysis and graph isomorphism), also rely on one type of features and thus might also be easily affected by compiler families and compilation settings. Furthermore, these methods are usually not as efficient as FLIRT due to the need for complex operations, e.g., graph isomorphism testing. In summary, none of the existing works offers an efficient, robust, and scalable solution for library function identification.

In this chapter, we aim to address the aforementioned challenges and limitations of existing works. Specifically, we focus on the following research problems:

- *How can we generate a "robust" signature for each library function that is resilient against compiler effects and obfuscation techniques?*
- *How can we rely on only those features whose extraction, indexing, and matching can be performed in an efficient manner?*
- *How can we design an efficient data structure to perform large-scale function matching (e.g., against millions of functions) relatively quickly (e.g., less than a second)?*

The main advantages of BINSHAPE are as follows. First, by relying mostly on lightweight features and the proposed data structure, our technique is *efficient*, and outperforms other techniques that rely on time-consuming computations, such as graph isomorphism. Second, incorporating different types of features significantly reduces the chance of signature collisions compared to most existing works which rely on a single type of features. Therefore, by extracting the aforementioned heterogeneous features and furthermore selecting the best features amongst them, our approach achieves a high level of *robustness*. Third,

our technique is not limited to a particular type of function, e.g., the wrapper functions provided by standard system libraries [114], and instead can identify a more general set of library functions. Finally, conducting the experimental evaluation on a large number (over a million) of functions confirms the *efficiency* and *scalability* of the proposed system.

Contribution. In summary, the main contributions of this chapter are as follows:

- *Extracting heterogeneous features:* To the best of our knowledge, this is the first effort in employing a diverse collection of features, including graph features, instruction-level characteristics, statistical characteristics, and function-call graphs, for library function identification.

- *Generating a robust signature:* The novel concept of function *shape* induces a single robust signature based on heterogeneous features. This choice allows our system to be accurate even when the code is subjected to slight modifications as the results of different compilers, compilation settings, and light obfuscation techniques.

- *Proposing a scalable technique:* By designing a novel data structure and using filters to prune the search space, our system demonstrates superior performance and provides a practical framework for large scale applications with millions of indexed library functions.

4.1.1 Motivating Example

Most of the existing works rely on a particular type of features, and they typically organize those features as a vector. In addition, for every version of the function, a new signature is generated and indexed in the repository. In this respect, our first observation is as follows: instead of using one type of features, the diverse nature of library functions demands a rich collection of features in order to increase the robustness of detection. In addition, as will be illustrated shortly in Figure 4.1, the most segregative features for different library

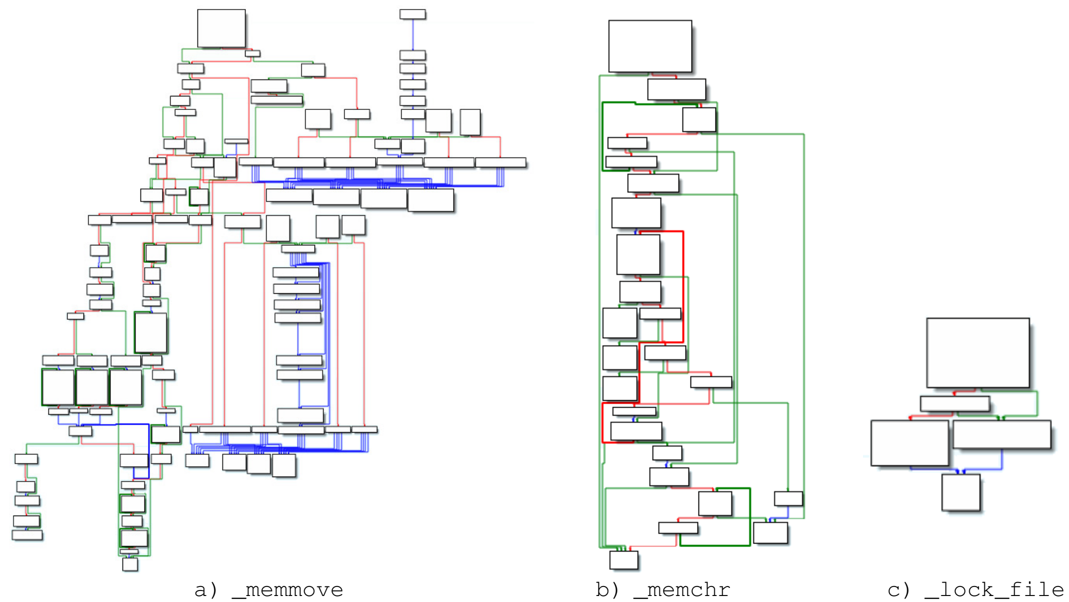


Figure 4.1: Control flow graphs of _memmove, _memchr, and _lock_file functions

functions will likely be different, and therefore a feature vector may not be the best way of representing a signature. To illustrate, let us consider the CFGs of the _memmove, _memchr, and _lock_file functions are depicted in Figure 4.1. We observe that two graph features of _memmove function are enough to make the functions distinguishable from others in our repository. On the other hand, the CFG of the _lock_file function contains a smaller number of nodes (i.e., five), and the CFG of the _memchr function is almost flat. Therefore, the best features to identify two different functions, one with few basic blocks and one with a large and complex CFG, would be very different; for instance, basic block level features for the former, and graph features for the latter.

4.1.2 Approach Overview

Our approach is divided into two phases: offline preparation (indexing) and online search (detection). As illustrated in the upper part of Figure 4.2, the offline preparation includes: (S) feature extraction discussed in Section 4.2, (1) feature selection presented in Section

4.2.5, which includes feature ranking to extract the elements of function *shape*, as well as the best feature selection; and (2) signature generation to index the functions in a repository, which is explained in Section 4.2.5. The lower part of Figure 4.2 depicts online search, which includes: (S) feature extraction explained in Section 4.2; (A) filtering process discussed in Section 4.3.2; and (B) detection technique presented in Section 4.3.

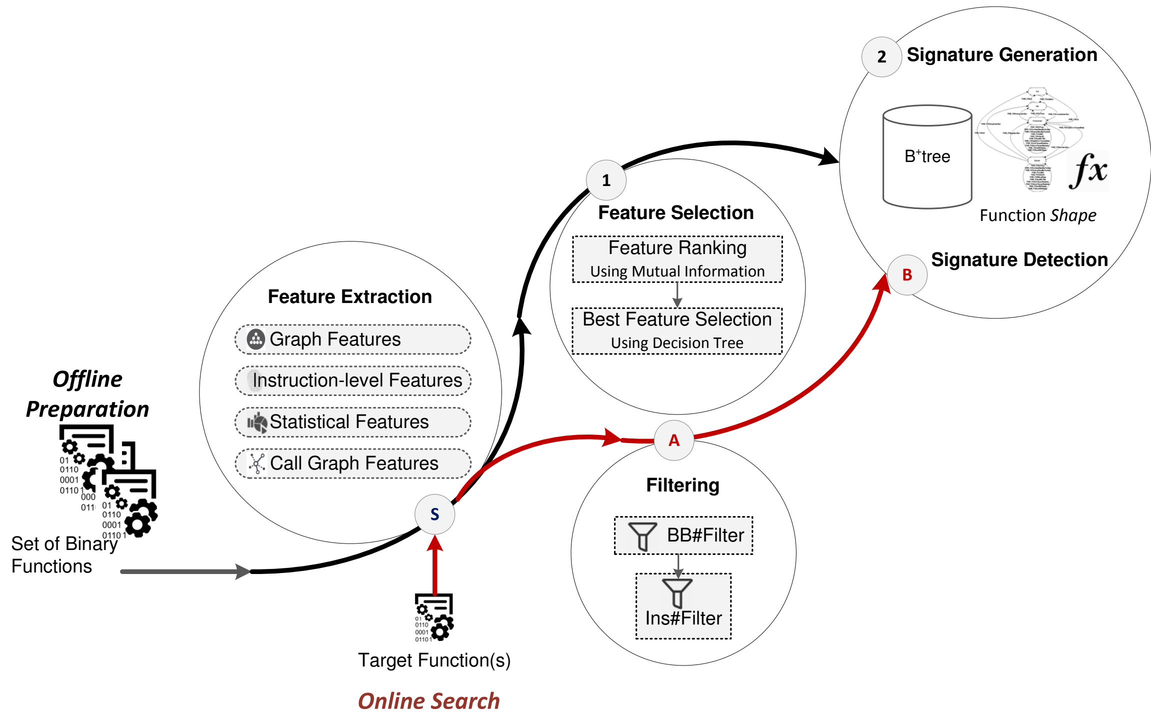


Figure 4.2: BINSHAPE approach overview

Initially, the binaries in our training set are disassembled using IDA PRO disassembler[1]. Subsequently, the graph features (Section 4.2.1) along with the instruction-level features (Section 4.2.2), statistical features (Section 4.2.3), and function-call graphs (Section 4.2.4) are extracted. To select the subsets of the features that are most useful to build the best signature, mutual information (Section 4.2.5) is employed on the extracted features. The top-ranked features are fed into a decision tree [88], and the outcome of the decision tree is stored in the proposed data structure (Section 4.3.1) to form a signature for each library

[1] https://www.hex-rays.com/products/ida/. Accessed on Dec 20, 2020.

function. In addition to such signatures, we also store the top-ranked features that compose the signature of each function. For detection, all the features are extracted from a given target binary. Two filters (Section 4.3.2) based on the number of basic blocks (`BB#`) and respectively the number of instructions (`Ins#`) are used to prune the search space. Consequently, a set of candidate functions are returned as the result of filtering, and finally the best matches are returned as the final results.

4.2 Feature Extraction

This section first describes different types of features, then presents the feature selection, and defines the so-called function *shape*.

4.2.1 Graph Feature Metrics

As the first category of the features, we consider the control flow graph of a binary function. To obtain the best features for each library function and to describe the *shape* of a function, we extract graph features based on different characteristics of the CFG. Among existing graph metrics [96], we only employ those which are inexpensive to extract. The selected graph features are listed in Table 4.1.

Example 4.2.1 *Below we show an example to illustrate the application of graph features to two functions. Our graph metrics are applied to two different library functions,* `memcpy_s` *and* `strcpy_s`*, as listed in Table 4.2. The corresponding CFGs have identical feature values for some metrics; for instance,* `numnodes`*,* `numedges`*, and* `cc` *are equal. In contrast, other metrics, such as* `graph_energy` *and* `pearson` *(shown in boldface) are different and can be used to discriminate the CFGs in this example (more generally, we will certainly need more features to uniquely characterize a function).*

Graph Metric	Description
n, numnodes	Number of nodes
e, numedges	Number of edges
p, num_conn_comp	Number of connected components
CC	Cyclomatic complexity: $e - n + 2p$
num_conn_triples	Number of connected triples
num_loop	Number of independent loops
leaf_nodes	Number of leaves
average_degree	$2 * e/n$
ave_path_length	Average distance between any two nodes
r, graph_radius	Minimum vertex eccentricity
link_density	$e/(n(n-1)/2)$
s_metric	Sum of products of degrees across all edges
rich_club_metric	Extent to which well-connected nodes also connect to each other
graph_energy	Sum of the absolute values of the real components of the eigenvalues
algeb_connectivity	2nd smallest eigenvalue of the Laplacian
pearson	Pearson coefficient for degree sequence of all edges
weighted_clust_coeff	Maximum value of the vector of node weighted clustering coefficients

Table 4.1: Examples of graph metrics

Graph Metric	`_memcpy_s`	`_strcpy_s`
n, numnodes	13	13
e, numedges	18	18
p, num_conn_comp	1	1
CC	7	7
num_conn_triples	6	6
num_loop	6	6
leaf_nodes	7	7
average_degree	2.7692	2.7692
ave_path_length	**2.2308**	**2.5**
r, graph_radius	**5**	**6**
link_density	0.2308	0.2308
s_metric	**150**	**159**
rich_club_metric	**0.2778**	**0.2778**
graph_energy	**18.7268**	**18.0511**
algeb_connectivity	**1**	**0.3820**
pearson	**0.4635**	**0.3415**
weighted_clust_coeff	**0.3334**	**0.5**

Table 4.2: Comparing graph features of `_memcpy_s` and `_strcpy_s`

As discussed before (Section 4.1.1), the graph features of `_memmove` function could be part of the best features, since these features can segregate `_memmove` function from

others. However, graph features alone are not sufficient, since there are cases where all the graph features of two different functions are identical, especially for functions of relatively small sizes. On the other hand, the CFG (and consequently the graph metrics) of a library function may differ due to the compilation settings or slight changes in the source file. Therefore, we consider additional features discussed in the following subsections.

4.2.2 Instruction-level Features

Instruction-level features carry both syntax and semantic information of a disassembled function. To this end, we propose to extract instruction-level features as a component of function *shape*. Some of the instruction-level features are listed in Table 4.3, such as the number of constants (`#constants`), and the number of callees (`#calls`).

Feature	**Description**	**Feature**	**Description**
`declaration`	declaration type	`instrnum`	number of instructions
`argsnum`	number of arguments	`numReg`	number of registers
`argsize`	size of arguments	`#mnemonics`	number of mnemonics
`localvarsize`	size of local variables	`#operand`	number of operands
`retType`	return type	`#calls`	number of callees
`constants`	constants	`#constants`	number of constants
`strings`	strings	`#strings`	number of strings

Table 4.3: Example of instruction-level features

In addition, inspired by [129], we categorize the instructions according to their operation types; for instance, we group the instructions that are related to stack operations (such as `push` and `pop`) into one category named `STK`. Our proposed categorisation (color classes) is presented in Table 4.4. By enriching standard CFGs with such information as different colors, there is a better chance to differentiate two functions even if they have the same CFG structure. This relates to the observation that these categories carry some information about the functionality of a program; for instance, encryption-related functions perform more logical and arithmetic operations rather than a function that writes to a file.

We record the frequency of each instruction category as a feature.

Mnemonic group	**Description**
`DTR`	Data transfer operations (e.g., `mov`)
`STK`	Stack operations (e.g., `push, pop`)
`CMP`	Compare operations (e.g., `cmp, test`)
`ATH`	Arithmetic operations (e.g., `add, sub`)
`LGC`	Logical operations (e.g., `and, or`)
`CTL`	Control transfer (e.g., `jmp, jne`)
`FLG`	Flag manipulation (e.g., `lahf, sahf`)
`FLT`	Float operations (e.g., `f2xm1, fabs`)
`CaLe`	System and interrupt operations (e.g., `sysexit`)

Table 4.4: Example of mnemonic groups

4.2.3 Statistical Features

Statistical analysis of binary code can be used to capture the semantics of a function. Several works have applied opcode analysis to binary code; for instance, opcode frequencies are used to detect metamorphic malware in [178, 202]. Therefore, each set of opcodes that belong to a specific function will likely follow a specific distribution according to the functionality they implement. For this purpose, we calculate the *skewness* and *kurtosis* measures to convert these distributions into scores as per equations 8 and 9 proposed in [170], as follows.

$$Sk = \left(\frac{\sqrt{N(N-1)}}{N-1}\right)\left(\frac{\sum_{i=1}^{N}(Y_i - \overline{Y})^3/N}{s^3}\right) \tag{8}$$

$$Kz = \frac{\sum_{i=1}^{N}(Y_i - \overline{Y})^4/N}{s^4} - 3 \tag{9}$$

where Y_i is the frequency of each instruction, $\overline{Y}$ is the mean, s is the standard deviation, and N is the number of data points. Similarly, we calculate the *z-score* [32] for each opcode (mnemonic), where the corpus includes all the functions in our repository.

Normalization. Each assembly instruction consists of a mnemonic and a sequence of up to three operands. The operands can be classified into three categories: memory references, registers, and constant values. We may have two fragments of a code that are both structurally and syntactically identical, but differ in terms of memory references, or registers [81]. Hence, it is essential that the assembly code be normalized prior to comparison. Therefore, the memory references and constant values are normalized to MEM and VAL, respectively. The registers can be generalized according to the various levels of normalization. The top-most level generalizes all registers regardless of types to REG. The next level differentiates General Registers (e.g., `eax`, `ebx`), Segment Registers (e.g., `cs`, `ds`), and Index and Pointer Registers (e.g., `esi`, `edi`). The third level breaks down the General Registers into three groups by size - namely, 32, 16, and 8-bit registers.

4.2.4 Function-Call Graph

Function-call graph is a structural representation that abstracts away instruction-level details, and can provide an approximation of a program functionality. Moreover, function-call graph is more resilient to instruction-level obfuscation that are usually employed by malware authors to evade the detection systems [93]. In addition, it offers a robust representation to detect different variants of the same malware programs [108]. Hence, the caller-callee relationship of the library functions is extracted.

The derived function-call graphs from those relationships are directed graphs containing a node corresponding to each function and edges representing calls from callers to callees. For labelling the nodes to exploit properties shared between functions, a neighbor hash graph kernel (NHGK) is applied to subsets of the call graph [104]. Those subsets include library functions and their neighbor functions (callees and callers). Function G maps the features of function f_i to a bit vector of length l, where l is the number of mnemonic categories (9 in total) as shown in Table 4.4. Function G checks each value of the mnemonic

groups of a given binary function; if the value is greater than 0, the corresponding bit vector is set to 1; otherwise, it will be 0.

$$G : f_i \rightarrow v_i = \{0, 1\}^l \tag{10}$$

The neighborhood hash value h for a function f_i and its set of neighbor functions N_{f_i} is computed by the following formula [93]:

$$h(f_i) = shr_1(G(f_i)) \oplus (\oplus_{f_j \in N_{f_i}} G(f_j)) \tag{11}$$

where shr_1 denotes a one-bit shift right operation and $\oplus$ indicates a bit-wise XOR. The time complexity of this computation is constant time, $O(ld)$, where d is the summation of outdegrees and indegrees, and l is the length of the bit vector. Therefore, we can represent a function-call graph with a hash value which preserves the relationship between the functions. For instance, suppose f_i is called by two other functions f_1 and f_2, and $f_i(i \notin \{1, 2, 3\})$ calls another function f_3. Therefore, the bit vectors based on the mnemonic group values of each function are generated (by the function G) to construct the set of neighbor function $N_{f_i} = \{v_1, v_2, v_3\}$. Finally, the hash value $h(f_i)$ would be equal to $shr_1(v_i) \oplus (v_1 \oplus v_2 \oplus v_3)$.

4.2.5 Feature Selection

After extracting all the aforementioned features, we end up with a number of features among which some might be the most relevant ones - those that appear more frequently and are most segregative in one function. Therefore, a feature selection process is conducted to reduce the number of features as well as to find the best ones. Our feature selection phase contains two major steps: *feature ranking* and *best feature selection* described as follows.

Feature Ranking

We measure the relevance of the aforementioned features based on the frequency of their appearance in each library function. Mutual information (MI) [173] represents the degree to which the uncertainty of knowing the value of a random variable is reduced given the value of another variable. To this end, we employ a Mutual Information measure to indicate the dependency degree between features X and library function labels Y as follows.

$$MI(X,Y) = \sum_{x \in X} \sum_{y \in Y} p(x,y) log_2 \frac{p(x,y)}{p(x)p(y)} \quad (12)$$

where x is the feature frequency, y is the class of library function (e.g., `memset`), $p(x,y)$ is the joint probability distribution function of x and y, and $p(x)$ and $p(y)$ are the marginal probability distribution functions of x and y. The main intention of feature ranking is to shorten the training time. We measure MI-based feature ranking on all categories of aforementioned features. In addition to MI, we apply other feature selection evaluators, such as *ChiSquared* [181], *GainRatio* [181], and *InfoGain* [183] using WEKA[2] [107]. Finally, the top-ranked features of our training dataset are selected based on the MI measure.

Best Feature Selection

Our aim is to build a classification system, which separates library functions from non-library functions. As such, we choose to apply a decision tree classifier on the top-ranked features obtained from the feature ranking process. Each library function is passed through the decision tree and the best provided features for that specific function are recorded. This automated task is performed on all library functions to create a signature for each function. For instance, the best features for `_strstr` function, in our dataset, are `#DTR`, `instrnum`, `algeb_connectivity`, `#constants`, and `average_degree` shown in Listing 4.1.

[2] https://www.cs.waikato.ac.nz/ml/weka/. Accessed on Dec 20, 2020.

Listing 4.1: `_strstr` best feature selection

```
#DTR > 32.500
| instrnum > 237: other {_strstr=0,other=69}
| instrnum ? 237: _strstr {_strstr=11,other=0}
#DTR ? 32.500
| algebraic_connectivity > 2.949
| | #constants > 3
| | | average_degree > 2.847: other {_strstr=0,other=591}
| | | average_degree ? 2.847
| | | |   instrnum > 171: _strstr {_strstr=1,other=0}
| | | |   instrnum ? 171: other {_strstr=0,other=48}
| | #constants ? 3: _strstr {_strstr=2,other=0}
| algebraic_connectivity ? 2.949:other{_strstr=0,other=4745}
```

4.3 Detection

Given a target binary program, we aim at matching the target disassembled functions to function signatures in our repository. The classical approach for detection would be to employ the closest Euclidean distances [77] between all top-ranked features extracted from the target function and the best features of the candidate functions from the repository. However, the best features vary for each candidate function and thus such an approach may not be scalable enough for handling millions of functions. Therefore, we design a novel data structure, called B^{++}tree, to efficiently organize the signatures of all the functions, and to find the best matches as explained in the following.

4.3.1 B^{++}tree Data Structure

Due to the growing number of free open-source libraries and the fact that binaries and malware are becoming bigger and more complex, indexing the signatures of a large number of library functions to enable efficient detection has thus become a demanding task. One classical approach is to store $(key, value)$ pairs as well as the indices of best features; however,

the time complexity of indexing and detection would be $O(n)$ and $O(mn)$, respectively, where n is the number of functions in the repository and m is the number of functions in a given binary.

To reduce the time complexity, we design a data structure, called B^{++}tree, that basically indexes the best feature values of all library functions in the repository in separate B^{+}trees, and links those B^{+}trees to corresponding features and functions. We also augment the B^{+}tree structure by adding *backward* and *forward* sibling pointers attached to each leaf node, which point to the previous and next leaf nodes, respectively. The number of neighbors is obtained by a user-defined *distance*. Consequently, slight changes in the values that might be due to the compiler effects or the slight changes in the source code is captured by the modified structure. Therefore, the indexing/detection time complexity will be reduced to $O(log(n))$ and $O(mlog(n))$, respectively. It is worth noting that the B^{+}tree could be replaced with similar data structures, such as red-black tree [53].

Example 4.3.1 *We explain the B^{++}tree structure with a small example illustrated in Figure 4.3. For each feature f_i in the list of top-ranked features $\{f_1, f_2, \ldots, f_m\}$, we generate a B^{+}tree. Then, the best features of all library functions F_i in our repository are indexed in the m number of B^{+}trees depicted in the middle box of Figure 4.3. For instance, if the best features of library function F_1 are f_1, f_2 and f_m, then, these three feature values linked to the function F_1 (shown in boldface) are indexed in the corresponding B^{+}trees. For the purpose of detection, (a) all the m features of a given target function are extracted. For each feature value, a lookup is performed on the corresponding B^{+}tree, and (b) a set of candidate functions based on the closest values and the user-defined distance are returned (we assume that $\{F_1, F_2, F_3, F_n\}$ are returned as the set of candidate functions). For instance, one match is found with the f_2 feature of function F_3 (shown in boldface in part b), whereas this feature is indexed as the best feature for F_1 function as well. Finally, the candidate functions are sorted based on the distance and total number of matches:*

$\{F_2, F_n, F_1, F_3\}$. *If we consider the first most frequent functions* ($t = 1$)*, then the final candidates would be* $\{F_2, F_n\}$ *functions.*

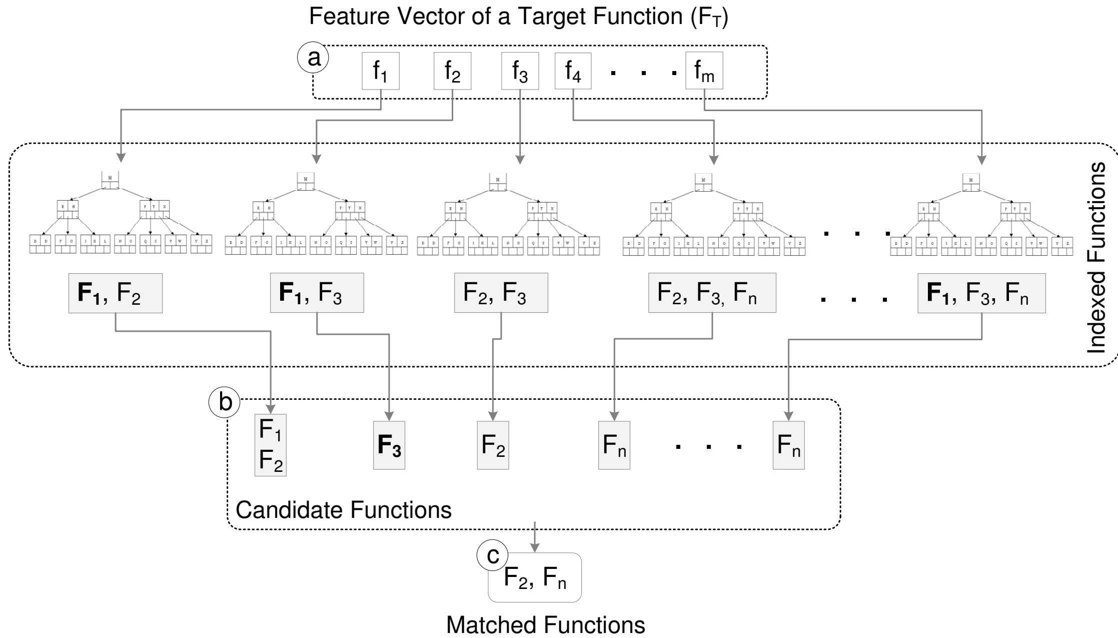


Figure 4.3: Indexing and detection structure

The details of the proposed detection approach are shown in Algorithm 2. Let f_T be the target function, with F retaining the set of candidate functions and their frequencies as output. First, all top-ranked features are extracted from the given target binary function f_T (line 7). By performing m (total number of features) lookups in each B$^+$tree (line 8), a set of candidate functions is returned (line 9). In order to choose the top t functions, the most t frequent functions are returned as the final set of matched functions (line 11).

4.3.2 Filtering

To address the scalability issue of dealing with large datasets, a filtering process is necessary. Instead of a pairwise comparison, we prune the search space by excluding functions that are unlikely to be matched.

Algorithm 2: Function Detection

```
    Input: f_T: Target function.
    Output: F: Set of candidate functions.
 1  m ← total number of features;
 2  n ← total number of functions in the repository;
 3  F_c ← {} ; dictionary of candidate functions ;
 4  t ← number of most frequent functions to be considered;
 5  distance ← user-defined distance;
 6  feature[m] ← array of size m to hold all the extracted features;
    begin
 7      feature[] = featureExtraction(f_T);
 8      foreach feature[i] ⊂ F_T do
 9          F_c = F_c + B+TreeLookup(feature[i], distance);
10      end
11      F = t_most_frequent_functions(F_c, t);
12      return F;
13  end
```

Basic Block Filter

It is unlikely that a function with four basic blocks can be matched to a function with 100 basic blocks. In addition, due to the compilation settings and various versions of the source code, there exist some differences in the number of basic blocks. Thus, a user-defined threshold value (γ) is employed, which should not be too small or too large to prevent discarding the correct matches. Therefore, given a target function f_T, the functions in the repository which have γ% more or less basic blocks than the f_T are considered as candidate functions for the final matching. Based on our experiments on our dataset, we consider the threshold value $\gamma = \pm 35$.

Instruction Filter

Similarly, given a target function f_T, the differences between the number of instructions of target function f_T and the functions in the repository are calculated; if the difference in the number of instructions is less than a user-defined threshold value λ, then the function is

considered as a candidate function. According to our dataset and performed experiments, we consider $\lambda = 35\%$.

4.4 Evaluation

In this section, we present the evaluation results of the proposed technique. First, we present the details of the experimental setup followed by the dataset description. Then, the main accuracy results of library function identification are presented. Furthermore, we study the impact of compilers on the proposed approach and discuss the results. Additionally, we examine the effect of feature selection on our accuracy results. We then evaluate the scalability of BINSHAPE on a large dataset. Finally, we study the effectiveness of BINSHAPE on a real malware samples.

4.4.1 Experimental Setup

We developed a proof-of-concept implementation in python to evaluate our technique. All of our experiments are conducted on machines running Windows 7 and Ubuntu 15.04 with Core i7 $3.4GHz$ CPU and $16GB$ RAM. The Matlab software has been used for the graph feature extraction. A subset of python scripts in the proposed system is used in tandem with IDA PRO disassembler. The MONGODB[3] database is utilized to store the proposed features for efficiency and scalability purposes. For the sake of usability, a graphical user interface in which binaries can be uploaded and analyzed is implemented. Any particular selection of data may not be representative of another selection. Hence, to mitigate the possibility that results may be biased by the particular choice of training and testing data, a `C4.5(J48)` decision tree is evaluated on a `90:10` training/test split of the dataset.

[3] https://www.mongodb.com/. Accessed on Dec 20, 2020.

4.4.2 Dataset Preparation

Program / Project	Version	Number of Functions	Size(Kb)
7zip/7z	15.14	133	1074
7zip/7z	15.11	133	1068
7-Zip/7zg	15.05 beta	3041	323
7-Zip/7zfm	15.05 beta	4901	476
expat	0.0.0.0	357	140
firefox	44.0	173095	37887
fltk	1.3.2	7587	2833
glew	1.5.1.0	563	306
jsoncpp	0.5.0	1056	13
lcms	8.0.920.14	668	182
libcurl	10.2.0.232	1456	427
libgd	1.3.0.27	883	497
libgmp	0.0.0.0	750	669
libjpeg	0.0.0.0	352	133
libpng	1.2.51	202	60
libpng	1.2.37	419	254
libssh2	0.12	429	115
libtheora	0.0.0.0	460	226
libtiff	3.6.1.1501	728	432
libxml2	27.3000.0.6	2815	1021
Notepad++	6.8.8	7796	2015
Notepad++	6.8.7	7768	2009
nspr	4.10.2.0	881	181
nss	27.0.1.5156	5979	1745
OpenSSL	0.9.8	1376	415
avgntopensslx	14.0.0.4576	3687	976
pcre3	3.9.0.0	52	48
python	3.5.1	1538	28070
python	2.7.1	358	18200
putty/putty	0.66 beta	1506	512
putty/plink	0.66 beta	1057	332
putty/pscp	0.66 beta	1157	344
putty/psftp	0.66 beta	1166	352
WireShark/Qt5Core	2.0.1	17723	3987
SQLite	2013	2498	1006
SQLite	2010	2462	965
SQLite	11.0.0.379	1252	307
tinyXML	2.0.2	533	147
Winedt	9.1	87	8617
WinMerge	2.14.0	405	6283
WireShark	2.0.1	70502	39658
WireShark/libjpeg	2.0.1	383	192
WireShark/libpng	2.0.1	509	171
xampp	5.6.15	5594	111436

Table 4.5: An excerpt of the projects included in our dataset

We evaluate our approach on a set of binaries, as detailed in Table 4.5. In order to create the ground truth, we download the source code of all C-library functions[4], as well as different versions of various open-source applications, such as `7-zip`. The source codes are compiled with Microsoft Visual Studio (*MSVC* 2010 and 2012), and GNU Compiler Collection (*GCC v4.1.2*) compilers, where the `/MT` and `-static` options, respectively, are set to statically link C/C++ libraries. In addition, the `O0-O3` options are used to examine the effects of optimization settings. Program debug databases (PDBs) holding debugging information are also generated for the ground truth. Furthermore, we obtain binaries and corresponding PDBs from their official websites (e.g., `WireShark`); for these binaries, their compilers are detected by a packer tool called EXEINFOPE[5]. Finally, the prepared dataset is used as the ground truth for our system, since we can verify our results by referring to the source code. In order to demonstrate the effectiveness of our approach to identify library functions in malware binaries, we additionally choose `Zeus` malware version 2.0.8.9, where the source code[6] was leaked in 2011 and is reused in our work.

4.4.3 Function Identification Accuracy Results

Our ultimate goal is to discover as many relevant functions as possible with less concern about false positives. Consequently, in our experiments we use the F-measure,

$$F_1 = 2 \times \frac{P \times R}{P + R} \quad , \quad Precision(P) = \frac{TP}{TP + FP} \quad , \quad Recall(R) = \frac{TP}{TP + FN} \qquad (13)$$

where TP indicates number of relevant functions that are correctly retrieved; FN presents the number of relevant functions that are not detected; FP indicates the number of irrelevant functions that are incorrectly detected; and TN returns the number of irrelevant functions that are not detected. To evaluate our system, we split the binaries in the ground truth

[4] `http://www.cplusplus.com/reference/clibrary/`. Accessed on Dec 20, 2020.
[5] `http://exeinfo.atwebpages.com/`. Accessed on Dec 20, 2020.
[6] `https://github.com/Visgean/Zeus`. Accessed on Dec 20, 2020.

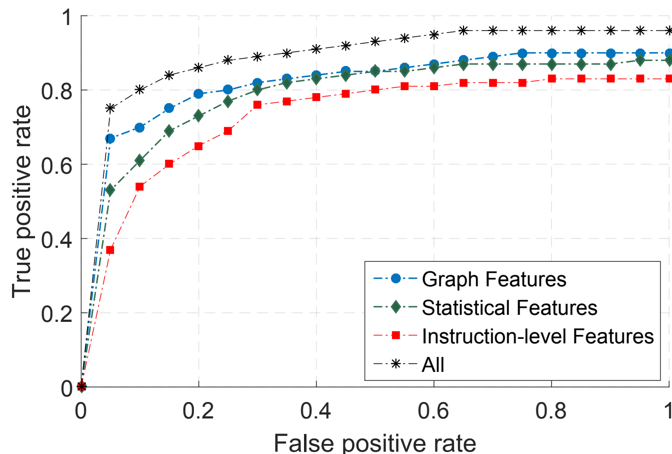


Figure 4.4: ROC curve for BINSHAPE features

into ten sets, reserving one as a testing set and using the remaining nine as training sets. We repeat this process 1000 times and report the results that are summarized in Figure 4.4. As seen, we obtain a slightly higher true positive rate when using graph features (including function-call graph feature) and statistical features. This small difference can be inferred due to the graph similarity between two library functions that are semantically close. Similarly, statistical features convey information related to the functionality of a function, which cause a slight higher accuracy. On the other hand, instruction-level features return lower true positive rate. However, when all the features are combined together, our system returns an average F_1 measure of 0.89.

4.4.4 Impact of Obfuscation

In the second scenario, we investigate the impact of obfuscation techniques on BINSHAPE as well as FLIRT and LIBV approaches. Our choices of obfuscation techniques are based on the popular obfuscator LLVM [120] and DALIN [140], which include *instruction substitution* (SUB), *bogus control flow* (BCF), *control flow flattening* (FLA), *register renaming* (RR), *instruction reordering* (IR), and *dead-code insertion* (DCI). *Instruction substitution*

replaces the instructions with functionally equivalent sequences of instructions. For instance, '`mov eax,edx`' can be substituted by '`push edx`' followed by '`pop eax`' instruction. *Bogus control flow* techniques insert new basic blocks that contain an opaque predicate and then make a conditional jump back to the original basic block. *Control flow flattening* flattens the control flow graph, which first splits the body of function into basic blocks and then puts all the blocks at same level (e.g., using a `switch` statement). *Register renaming*, renames registers that are utilized for the same purpose. *Instruction reordering* reorders independent instructions in which the functionality remains intact while the syntax changes. For instance, the taken (True) and not taken (False) branches are reversed and the conditional branch instruction is inverted. *Dead-code insertion* injects additional instructions, which are not executed or have to effect on the functionality of a program. Dead-code insertion can be accomplished by mathematical operations (e.g., `add` and `sub`) or jump instructions.

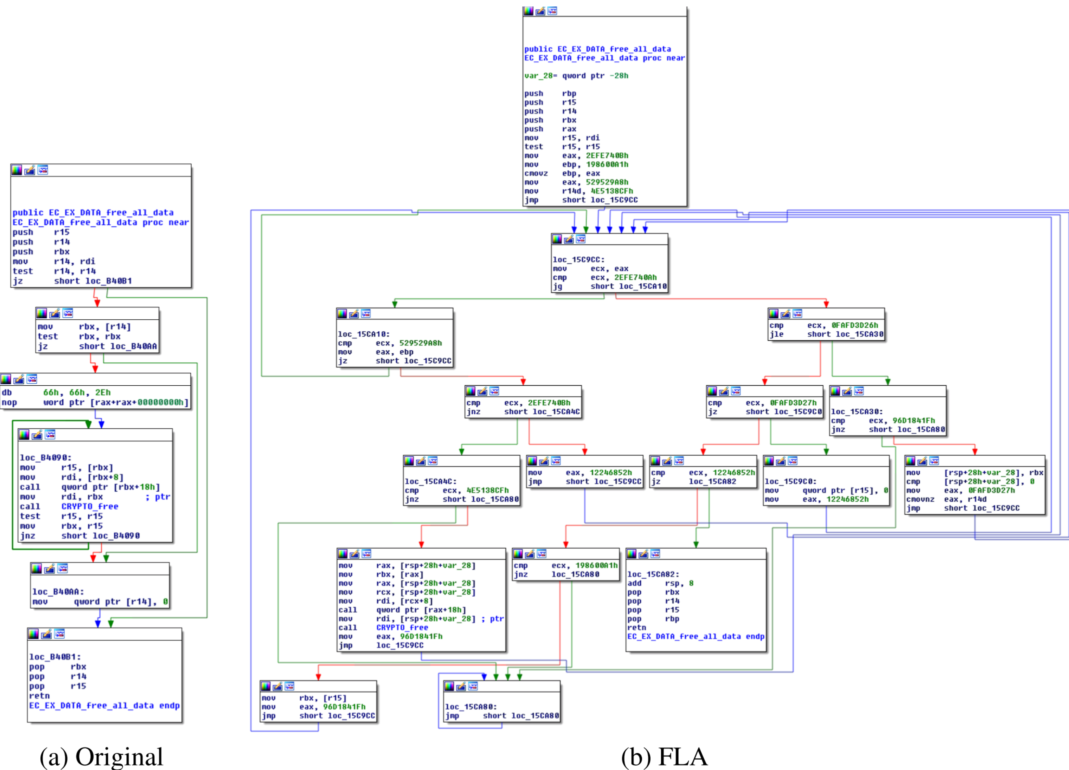
In this set of experiments, we collect a random set of samples (i.e., 25) that are not obfuscated, test our model and report accuracy results. The obfuscation is then applied, in which obfuscated binaries are tested against original binaries and new accuracy measurements are obtained. The effects of obfuscation tehcniques are presented in Table 4.6. As shown, BINSHAPE can overcome some obfuscation effects. The accuracy remains the same when RR and IR techniques are applied, while it is reduced slightly in the case of DCI and SUB obfuscations. The reason is that most of the features which are not extracted at the instruction level (e.g., graph features), are not significantly affected by these techniques. In addition, normalizing the assembly instructions eliminates the effects of RR, whereas, statistical features are more affected by DCI and SUB techniques, since these features rely on the instructions frequencies.

However, the accuracy results after applying FLA and BCF through LLVM obfuscator

Obfuscation	BINSHAPE ($F_1 = 0.89$)	FLIRT ($F_1 = 0.81$)	LIBV ($F_1 = 0.84$)
Technique	F_1^*	F_1^*	F_1^*
RR	0.89	0.81	0.84
IR	0.89	0.78	0.82
DCI	0.88	0.80	0.82
SUB	0.86	0.79	0.80
All	0.86	0.76	0.80

Table 4.6: F-measure before/after (F_1/F_1^*) applying obfuscation

are not promising, and we exclude them from our experimental results. We show the reason behind such results with an illustrative example. The CFGs of original and obfuscated (e.g., FLA) versions of `EC_EX_DATA_free_all_data` function from `OpenSSL` are illustrated in Figure 4.5a and Figure 4.5b, respectively. As shown, the *shape* of the obfuscated function is totally different than its original version, which indicates that the best



(a) Original (b) FLA

Figure 4.5: CFGs of `EC_EX_DATA_free_all_data` function in `OpenSSL` - libssl

features of the original function high likely would not match with the features of FLA version. Therefore, additional de-obfuscation techniques are required for handling these kinds of obfuscations. However, one of the future directions of this work is to perform statistical analysis to identify obfuscation-resilient features, which remain almost the same in the presence of various obfuscation techniques. Based on our manual analysis, this set of obfuscation-resilient features might include number of calls, system calls and constants.

4.4.5 Impact of Compilers

In this section, we examine the effects of compilers on a random subset of binaries in different scenarios as follows. (i) *The impact of compiler version*: We train our system with binaries compiled with *MSVC* 2010 at optimization level `O2`, and test it with binaries compiled with *MSVC* 2012 with the same optimization level. (ii) *The impact of optimization levels*: We train our system with binaries compiled with *MSVC* 2010 at optimization level `O1`, and test it with the same compiler at optimization level `O2`. (iii) *The impact of different compilers*: We collect binaries compiled with *MSVC* 2010 and optimization level `O2` as training dataset, and test the system with binaries compiled with *GCC* v4.1.2 compiler at optimization level `O2`. The obtained precision and recall for the aforementioned scenarios are reported in Table 4.7. We observe that our system is not affected significantly by changing either the compiler versions or the optimization levels. This can be interpreted due to the libraries and robust features that are selected during the feature ranking and best feature selection processes. However, different compilers affect the accuracy.

4.4.6 Impact of Feature Selection

We carry out a set of experiments to measure the impact of feature selection process, including top-ranked feature selection as well as best feature selection. First, we test our system to determine the best threshold value for top-ranked features as shown in Figure 4.6. We start

Project	Version		Optimization		Compiler	
	MSVC.2010 vs *MSVC*.2012 O2		*MSVC*.2010 O1 vs O2		*MSVC*.2010 vs GCC-4.1.2 O2	
	Precision	**Recall**	**Precision**	**Recall**	**Precision**	**Recall**
`bzip2`	1.00	0.98	0.90	0.85	0.82	0.80
`OpenSSL`	0.93	0.78	0.91	0.80	0.83	0.78
`Notepad++`	0.98	0.97	0.95	0.82	0.84	0.72
`libpng`	1.00	1.00	0.91	0.74	0.81	0.72
`TestSTL`	0.98	1.00	0.90	0.84	0.81	0.75
`libjpeg`	0.93	0.90	0.88	0.76	0.81	0.69
`SQLite`	0.91	0.87	0.89	0.85	0.78	0.71
`tinyXML`	1.00	0.99	0.90	0.82	0.84	0.79

Table 4.7: Impact of compiler versions, optimization settings, and compilers families

by considering five top-ranked features and report the F_1 measure of 0.71. We increment the number of top-ranked features by five each time. When the number of top-ranked features reaches 35 classes, the F_1 measure is increased to 0.89 and it remains almost constant afterwards. Based on our findings, we choose 35 as the threshold value for the top-ranked feature classes.

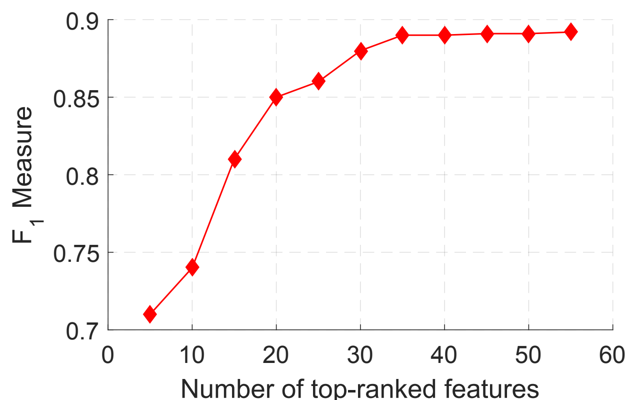


Figure 4.6: Impact of top-ranked features

Next, we pass the top-ranked features into the decision tree in order to select the best features for each function. The goal is to investigate whether considering the subset of best features would be enough to segregate the functions. In order to examine the effect of best features, we perform a breadth first search (BFS) on the corresponding trees to sort

best features based on their importance in the function; since the closer the feature is to the root, the more it is segregative. Our experiments examine the F_1 measure while varying the percentage of best features. We start by 40% of the top-ranked best features and increment them by 10% each time. Figure 4.7 shows the relationship between the percentage of best features and the F_1 measure. Based on our experiments, we find that 90% of the best features results in an F_1 measure of 0.89. However, for the sake of simplicity, we consider all the selected best features in our experiments.

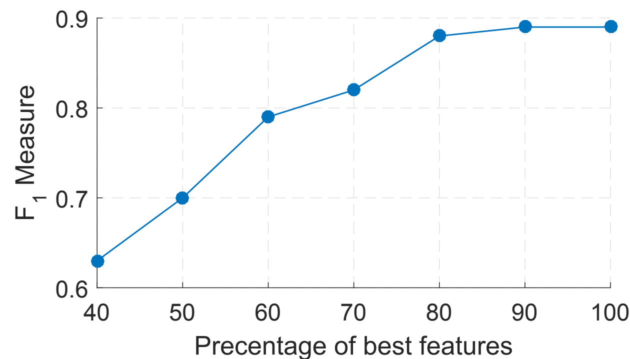


Figure 4.7: Impact of best features

4.4.7 Impact of Filtering

We study the impact of the proposed filters (e.g., `BB#` and `Ins#`) on the accuracy of BINSHAPE. For this purpose, we perform four experiments by applying: (i) no filter (ii) `BB#` filter, (iii) `Ins#` filter, and (iv) both of the aforementioned two filters. As shown in Figure 4.8, the drop in accuracy caused by the proposed filters is negligible. For instance, when we test our system with two filters, the highest drop in accuracy is about 0.017.

4.4.8 Scalability Study

To evaluate the scalability of our system, we prepare a large collection of binaries consisting of different '.exe' or '.dll' files (e.g., `msvcr100.dll`) containing more than $3,020,000$ disassembled functions. We gradually index this collection of functions in a random order,

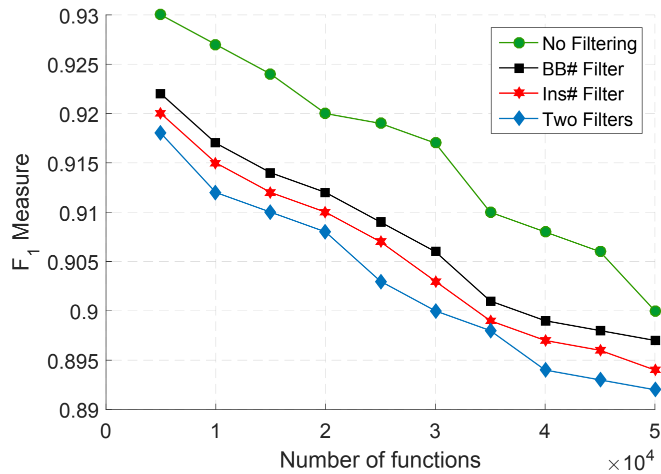


Figure 4.8: Impact of filtering

and query the `7-zip` binary file of version 15.14 on our system at an indexing interval of every 500,000 assembly functions. We collect the average indexing time for each function to be indexed, as well as the average time it takes to respond to a function detection. The indexing time includes feature extraction and storing them in the B$^+$trees. Figure 4.9 depicts the average indexing and detection time for each function.

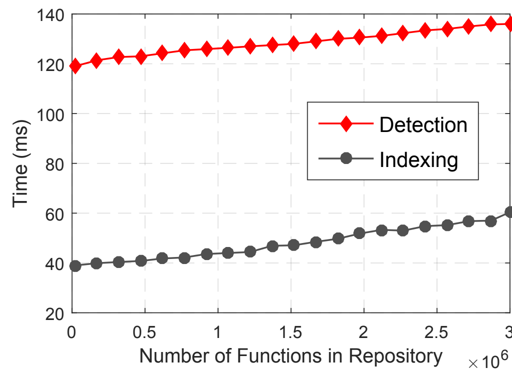


Figure 4.9: Scalability study

The results suggest that our system scales well with respect to the repository size. When the number of functions increases from 500,000 to 3,020,000, the impact on response time of our system is negligible (0.14 seconds on average to detect a function amongst three million functions in the repository). We notice through our experiments that the ranking

time and filtering time are very small and negligible. For instance, ranking $5,000$ features takes $0.0003\ ms$, and filtering highly likely similar functions in the repository to a function having 100 basic blocks and $10,000$ instructions, takes $0.009\ ms$. Besides, the feature extraction time varies for different types of features; e.g., the graph feature extraction takes more time than instruction-level and call-graph feature extractions.

4.4.9 Application to Malware

We are further interested in studying the applicability of our approach in identifying library functions in malware binaries. However, one challenge is the lack of ground truth to verify the results due to the nature of malware. Consequently, we consider `Zeus` version 2.0.8.9, where the leaked source code is available. First, we compile the source code with *MSVC* and *GCC* compilers, and keep the debug information for the purpose of verification. Second, we compile `UltraVNC`, `info-zip`, `xterm` and `BEAEngine` libraries with *MSVC* and *GCC* compilers and index them into our repository. We choose the aforementioned libraries based on the technical report [124] that reveals which software components are reused by `Zeus`. Finally, we test the compiled binaries of `Zeus` to find the similar functions with the functions in our repository. By manually examining the source code as well as the debug information at binary level, we are able to verify the results listed in Table 4.8. We observe through our experiments that the statistical features as well as graph features are the most powerful features in discovering free open-source library functions.

Library	Number of Functions	BinShape	
		Found	FP
`UltraVNC`	20	28	11
`info-zip`	30	27	0
`xterm`	17	18	2
`BEAEngine`	21	20	0

Table 4.8: Function identification in `Zeus`

4.5 Limitations and Concluding Remarks

In this chapter, we presented a pioneering investigation into the possibility of representing a function based on its *shape*. We proposed a robust signature for each library function based on a diverse collection of heterogeneous features, covering CFGs, instruction-level characteristics, statistical features, and function-call graphs. In addition, we designed a novel data structure, which includes B^+tree, to efficiently support accurate and scalable detection. Experimental results demonstrated the effectiveness of BINSHAPE. The main advantages of BINSHAPE are as follows. First, by relying mostly on light-weight features and the proposed data structure, it is *efficient*, and outperforms techniques that rely on time-consuming computations such as graph isomorphism. Second, incorporating different types of features significantly reduces the chance of signature collisions compared to most existing works which rely on a single type of features. Therefore, by extracting the aforementioned heterogeneous features and furthermore selecting the best features amongst them, BINSHAPE achieves a great deal of *robustness*. Third, the proposed technique is general in the sense that it is not limited to a particular type of functions, e.g., the wrapper functions provided by standard system libraries [114]. Finally, testing against a large number (over a million) of functions in a repository confirms the *efficiency* and *scalability* of BINSHAPE.

However, the proposed approach has the following limitations: (i) We have not scrutinized the impact of inline functions. (ii) Our system is able to tackle some code transformation, such as instruction reordering, however, some other obfuscation techniques (e.g., control flow flattening) affect the accuracy of BINSHAPE; similar to existing solutions, BINSHAPE is not designed to handle packed, encrypted, and obfuscated binaries. (iii) We have not examined the effects of other compilers, such as *icc*, on our approach. (iv) We have not investigated the impact of CPU architectures, such as MIPS or ARM in this study. Several of these limitations will be addressed in the following chapters and the rests will be considered as potential future work.

Chapter 5

Vulnerable Library Function Detection in Binaries and Firmware

With the wide-adoption of digital technologies, the security concerns of their underlying software or firmware become critically important. Hence, we aim at applying binary analysis on the firmware images of the Internet of Things (IoT) devices. To this end, we focus on the smart grid domain, one of the major critical infrastructures that generates and distributes power and electric energy. There is a widespread adoption of intelligent electronic devices (IEDs) in modern-day smart grid deployments. Consequently, any vulnerabilities in IED firmware might greatly affect the security and functionality of the smart grid. In this chapter, we present BINARM, a scalable approach based on a multi-stage fuzzy matching engine to detecting vulnerable functions in IED firmware mainly for the ARM architecture.

This chapter is organized as follows. First, the vulnerable function detection problem and approach overview are discussed in Sections 5.1 and 5.2, respectively. Then, the IED firmware and vulnerability databases generations are presented in Sections 5.3. Next, the BINARM's multi-stage detection engine is explained in Section 5.4. Afterwards, the evaluation resutls are presented in Section 5.5. Finally, the BINARM's limitations along with the concluding remarks and related future research directions are presented in Section 5.6.

5.1 Introduction

Smart grid is a network of transmission lines, substations, transformers, and distribution systems that deliver electricity from the generation units to residences and businesses customers. The digital technology provides the two-way communication between the utility and its customers, and the sensing along the transmission lines. Intelligent electronic devices (IEDs) play an important role in typical smart grids by supporting supervisory control and data acquisition (SCADA) communications, condition-based monitoring, and polling for event-specific data in the substations. Cyberattacks on smart grids deployment could lead to infrastructural failure, blackouts, energy theft, customer privacy breach, etc. The firmware (software) running on IEDs is subject to a wide range of software vulnerabilities, and consequently cyberattacks exploiting such security vulnerabilities may have debilitating repercussions on national economic security and national safety[1].

In fact, a startling increase in the number of attacks against industrial control systems (ICS) has been observed (e.g., a 110% increase when comparing 2016 to 2015[2]. A prime example of such an attack is *Industroyer* [48, 111] targeting Ukraine's power grid, which is capable of directly controlling substation switches and circuit breakers. As other examples, the *BlackEnergy* [164] APT took control of operators' control stations and utilized them to cause a blackout, while *Stuxnet* [80, 134] targeted Siemens ICS equipment in order to infiltrate Iranian nuclear facilities. Besides those real-world attacks, recent analysis demonstrates similar threats in other countries, e.g., with 50 power generators taken over by attackers, as many as 93 million US residents may be left without power [188]. These real-world attacks or hypothetical scenarios indicate a clear potential and serious consequences for future attacks against critical infrastructures including smart grids.

[1] https://www.dhs.gov/critical-infrastructure-sectors. Accessed on Dec 20, 2020.

[2] https://securityintelligence.com/attacks-targeting-industrial-control-systems-ics-up-110-percent/. Accessed on Dec 20, 2020.

Identifying security-critical vulnerabilities in firmware images running on IEDs is essential to assess the security of a smart grid. However, this task is especially challenging, since the source code of firmware is usually not available. In the literature, general-purpose techniques have been developed to automatically identify vulnerabilities in embedded firmware based on dynamic analysis (e.g., [44, 64, 192, 217]) or static analysis and code similarity approaches (e.g., [54, 78, 84, 174, 213]). To the best of our knowledge, none of the existing works focuses on the smart grid context. Although such general-purpose techniques are also applicable to the firmware of smart grid IEDs, they share some common limitations as follows:

(i) *Applicability*: They lack sufficient domain knowledge specific to smart grid IEDs, such as a database of known vulnerabilities in such IEDs as well as their firmware images. Therefore, in the existing works: (a) no prior knowledge about the scope (e.g., smart grid) is required; (b) no additional effort to gather and analyse the relevant IED firmware images is needed. They can easily download any type of firmware from the wild; and (c) no study on the reused open-source libraries in the IED firmware images is performed; it is highly likely that most relevant libraries are not included in their vulnerability dataset, which might result in higher false negative rates.

(ii) *Scalability*: Some of the existing approaches typically rely on expensive operations, such as semantic hashing [174], and they typically lack effective filtering steps to speed up the function matching. Consequently, those techniques are usually not efficient enough to handle the much larger sizes of IED firmware images (e.g., compared to that of network routers).

(iii) *Adaptability*: Handling the presence of a new common vulnerabilities and exposures (CVE)[3] and efficiently indexing those new vulnerabilities poses another challenge to some existing works (e.g., [213]).

[3] https://nvd.nist.gov/. Accessed on Dec 20, 2020.

In this chapter, we present BINARM, a scalable approach to detecting open-source library functions as well as vulnerable functions in smart grid IED firmware based on the ARM architecture. To this end, we first build a large-scale vulnerability database consisting of common vulnerabilities in IED firmware images. The design of our vulnerability database is highly influenced and guided by the prominent open-source libraries that are reused in the IED firmware images. Identifying these IEDs and obtaining the corresponding firmware images require significant efforts as follows: (i) identify relevant manufacturers; (ii) collect and study the corresponding IED firmware images; (iii) identify the used open-source libraries in these images; and (iv) cross-reference the identified open-source libraries with the CVE database[4] and compile the list of CVEs.

Second, in order to ensure BINARM that is efficient and scalable enough to handle IED firmware images, we design a detection engine that employs three increasingly complex stages in order to speed up the process by filtering irrelevant candidates as early as possible. Third, BINARM does not only provide a similarity score as prior efforts, such as [78, 213], but also presents in-depth analysis (at instruction level, basic block level, and function level) to justify the results of the matching and to assist reverse engineers for further investigation. We conduct extensive experiments with a large number of real-world smart grid IED firmware from various vendors in order to evaluate the effectiveness and scalability of BINARM.

Contributions. The main contributions of this chapter are as follows:

- To the best of our knowledge, we develop the first large-scale vulnerability database specifically for IEDs firmware covering most of the major vendors. In addition, we build the first IED firmware database, which provides an overview of the state of the industry. Such effort can be leveraged for future research on smart grid IEDs, and can be beneficial to IED vendors and utilities to assess the security of the IED firmware.

[4] https://cve.mitre.org/. Accessed on Dec 20, 2020.

- We propose a multi-stage detection engine to efficiently identify vulnerable functions in IED firmware, while maintaining high accuracy. The experimental results demonstrate the efficacy of our system such that the engine is three orders of magnitude faster than the existing fuzzy matching approach [110].

- Our experimental results ascertain the accuracy of the proposed system, with an average accuracy of 0.92. In addition, the applicability of BINARM is confirmed in our study; it successfully detects 93 potential CVEs in real-world IED firmware images within 0.09 seconds per function on average, the majority of which have been confirmed by our manual analysis.

5.2 Approach Overview

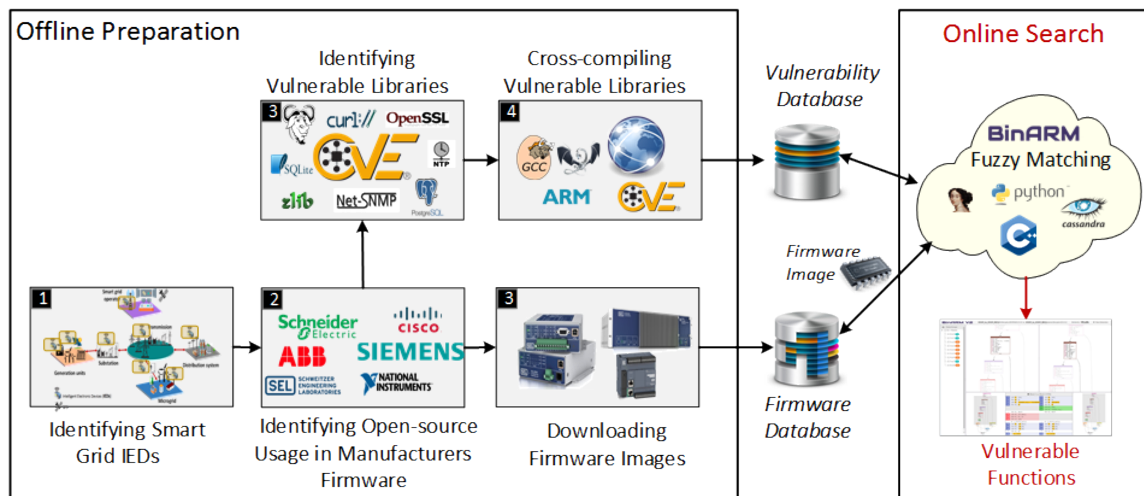


Figure 5.1: BINARM overview

An overview of our approach is depicted in Figure 5.1, which consists of two major phases: the *offline preparation* and *online search*. The *offline preparation* phase consists in the creation of two comprehensive databases; one containing a set of IED firmware and the other known vulnerabilities specific to IEDs. To this end, we carry out the following:

- Identify the IEDs and a set of manufacturers that provide equipment for smart grids.
- Collect relevant IED firmware produced by the identified manufacturers, and store the images in the *Firmware Database*.

Such information further provides insight about which libraries might be utilized by each manufacturer in their released firmware, which enables us to build our vulnerability database. For this purpose, we process as follows:

- Determine reused libraries in the IED firmware from manufacturers' websites or available documentations, e.g, the copyright provided by NI[5].
- Collect identified open-source libraries, compile them for the ARM architecture, and finally cross-reference with CVE database to build the *Vulnerability Database*.

During the *online search* phase, BINARM assesses the security of a new or previously-indexed IED firmware by matching against known vulnerable functions in our *Vulnerability* database. To efficiently deal with a large number of functions, we design a multi-stage fuzzy matching detection engine. The proposed engine gradually excludes candidate functions from the analysis, where these function are less likely similar to the target function. The similarity results provide an in-depth analysis that contains the similarity scores of the two functions at the function level (e,g., CFG), the basic-block level, and the instruction level, all of which are presented within a graphical user interface. Hence, the analyst can dig into the similarity results and gain more insights about the matched function.

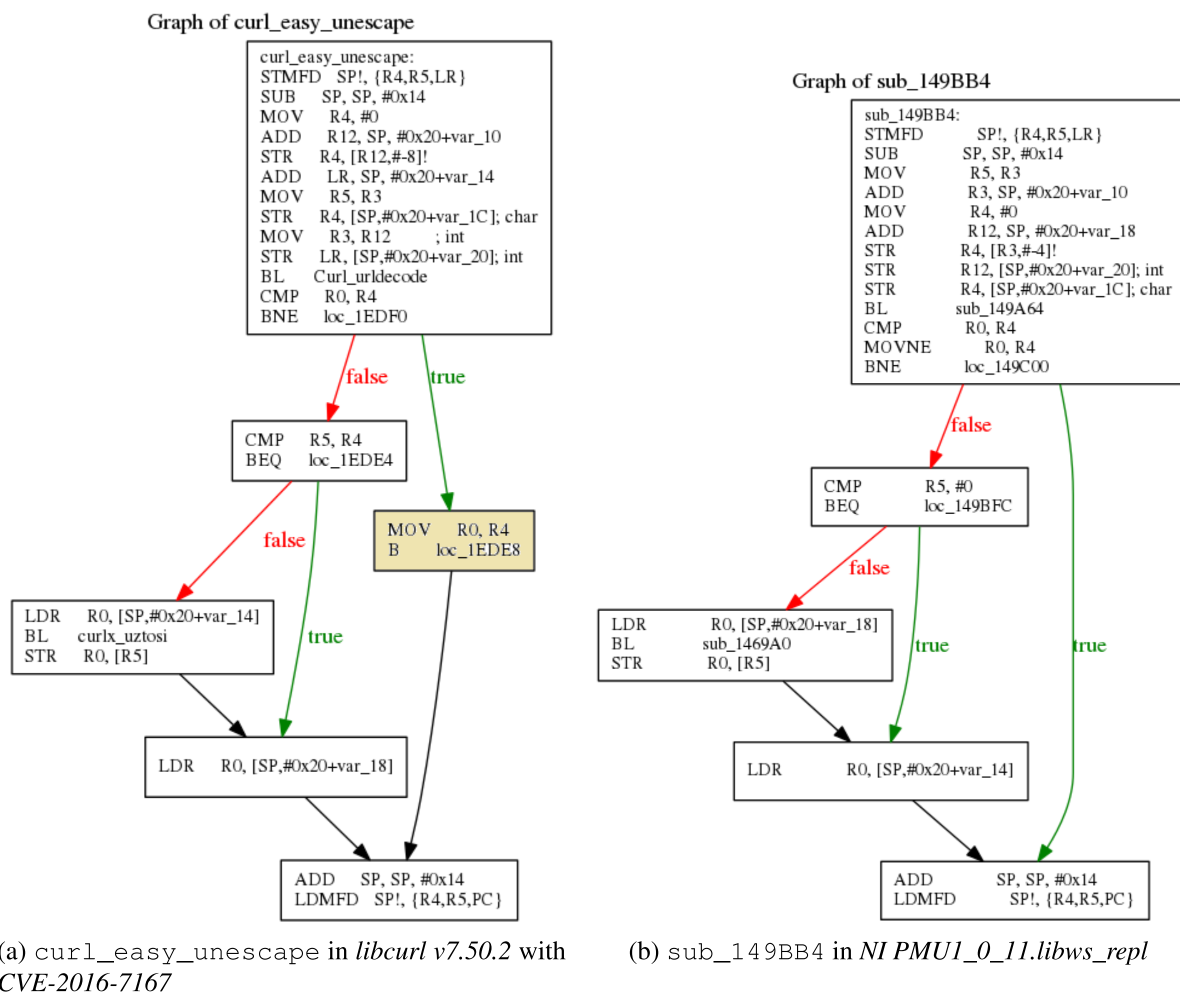
We demonstrate how the aforementioned process works by applying it to the following motivating example. Suppose a fictitious utility company would like to deploy several phasor measurement units (PMUs) and is concerned about potential vulnerabilities inside those units. Following our methodology depicted in Figure 5.1, we would first identify

[5] http://zone.ni.com/reference/en-XX/help/374498F-01/insightcm/bp_copyright/. Accessed on Jan 15, 2018.

the manufacturer, e.g., given by the utility as National Instruments (NI) in this particular example. Second, we would collect the IED firmware, which is given by the utility as *NI PMU1_0_11* firmware image[6]. Third, we would identify the reused libraries in this firmware, e.g., the `libcurl` v7.50.2 library. Fourth, we would identify vulnerable functions inside each library, e.g., a vulnerable function inside the `libcurl` v7.50.2 library as depicted in Figure 5.2a. Finally, we employ our detection engine to find matching functions in the provided firmware image, e.g., a matched function shown in Figure 5.2b. As shown, the two functions have a high degree of similarity; indeed, the main difference is the presence of an additional basic block consisting of two instructions (highlighted in Figure 5.2a) in the `curl_easy_unescape` function. This similarity implies that the function in Figure 5.2b may also have the CVE-2016-7167 vulnerability, which provides useful information for the utility to take corresponding actions.

We note that, although this particular example may make it seem relatively straightforward to detect vulnerable functions in a firmware, this is usually not the case in practice due to two main challenges. First, the needed information about manufacturer, libraries, and vulnerabilities may not be readily available from the utility company as in this example. For this reason, we will build our vulnerability and firmware databases, as discussed in Section 5.3. Second, the function matching process may be too expensive for utility companies, since they may be dealing with the constant deployment or upgrade of a large number of IEDs from different manufacturers. Cross checking such a large number of firmware images with an even larger number of library functions (e.g., $5,103$ vulnerable functions) can take significant effort. For instance, the *Linksys WRT32X* firmware image with $39kb$ size contains $47,025$ functions, whereas the *NI PMU1_0_11* firmware comprises $226,496$ functions and is $256kb$ large. To address this challenge, we propose our efficient multi-stage detection engine in Section 5.4.

[6] http://digital.ni.com/public.nsf/allkb/5391E8424944D0BC86257E45000B025C. Accessed on Jan 15, 2018.



(a) `curl_easy_unescape` in *libcurl v7.50.2* with *CVE-2016-7167*

(b) `sub_149BB4` in *NI PMU1_0_11.libws_repl*

Figure 5.2: Comparing the CFGs of a vulnerable function and an unknown function

5.3 Building IED Firmware and Vulnerability Databases

Identifying the IEDs and obtaining their corresponding firmware can help vendors and utilities in assessing the security of elaborated or deployed IEDs firmware. However, this process requires significantly more effort than simply acquiring firmware from any consumer devices by crawling and downloading from the wild. In this section, we provide the background of smart grid IEDs, and then elaborate on the creation and content of our firmware and vulnerability databases.

5.3.1 Intelligent Electronic Devices in the Smart Grid

A power grid is a complex and critical system to provide generated power to a diverse set of end users. It is composed of three main sectors: generation, transmission, and distribution. An overview of the smart grid is illustrated in Figure 5.3. The transmission system takes the power that is generated by the generation system and delivers it to distribution substations. The role of a distribution substation is to transform received high voltage electricity to a lower more suitable voltage for distribution among customers. With the introduction of IEC 61850[7] standard, technologies such as Ethernet, high-speed wide area networks (WANs), and powerful but cost-effective computers are leveraged in order to define a modern architecture for communication within a substation [148]. Consequently, a vast set of devices, labeled as intelligent electronic devices (IEDs), are emerged into the smart grid infrastructure. Such devices are coupled with traditional ICS and power equipment which enables their integration into the network.

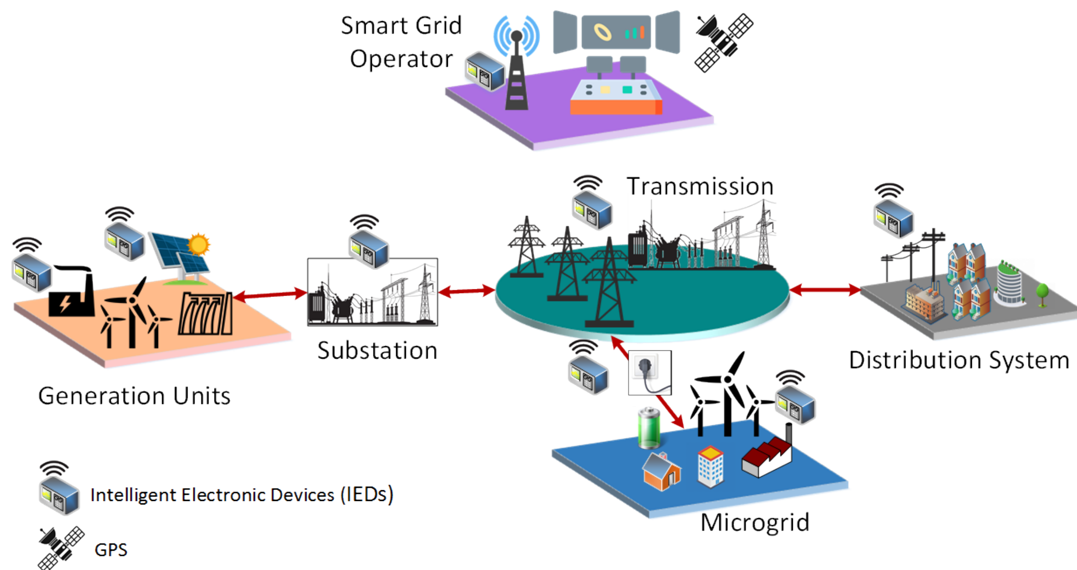


Figure 5.3: Smart grid overview

[7] https://webstore.iec.ch/publication/6028. Accessed on Dec 20, 2020.

Four different applications in a smart grid benefit from the extensive use of IEDs as shown in Figure 5.4, including:

(i) *Control applications*: send and receive commands to control the system behaviour remotely, such as load-shedding, power system stabilizers, and voltage regulators;

(ii) *Monitoring*: convert received analog input (e.g., currents, voltages, power values) from primary equipment into a digital format and leverage it to evaluate the performance of the system or detect operation issues. Some of the examples include phasor measurement units (PMUs), advanced metering infrastructures (AMIs), and phasor data concentrators (PDCs);

(iii) *Protection*: detect faults that need to be isolated from the network in a specific and timely manner, such as reclosers and various types of relays; and

(iv) *Communication*: transmit and receive the data, measurements, and time signals that are required for the operation of the other three applications, such as GPS antenna, switches, and gateways.

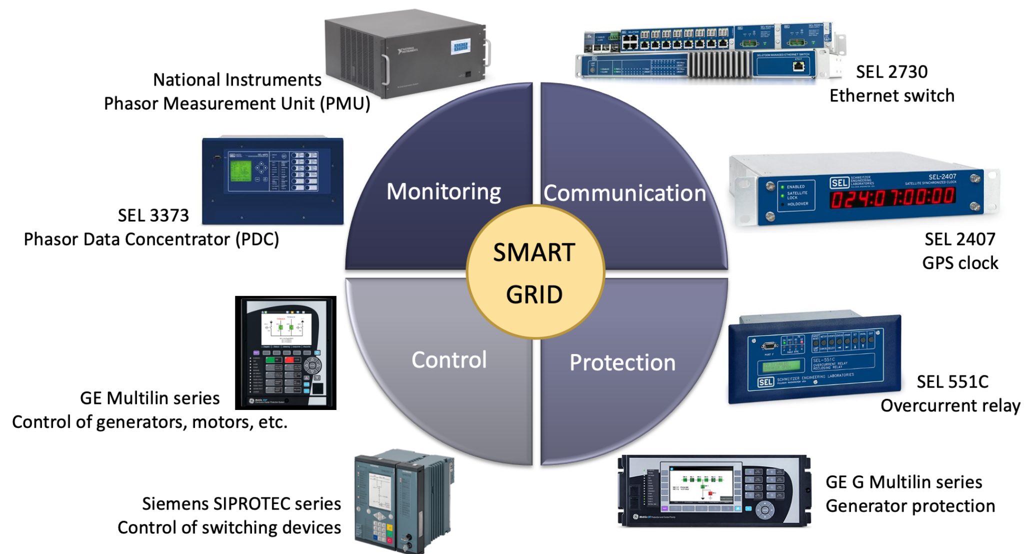


Figure 5.4: Four applications in the smart grid and some examples of related IEDs

5.3.2 Manufacturer Identification

In order to identify a set of relevant manufacturers and market dynamics, we first reduce the scope to the substations and the networking equipment that allows the smart grid to operate. Then, we study the categorization of vendors in the smart grid ecosystem by using different sources, such as *GTM Research*[8] and *Cleantech Group* [165] reports. This information provides the necessary insights in order to identify top smart grid manufacturers, as listed in Table 5.1. Such knowledge becomes the foundation to further determine relevant libraries, vulnerabilities and IED firmware images.

Manufacturer	Relevant Component(s)
ABB Schweiz AG	Automation Hardware
National Instruments	Automation Hardware
Schneider Electric	Automation Hardware
Schweitzer Engineering Laboratories (SEL)	Automation Hardware
Siemens	Automation Hardware
Elster (Honeywell)	Automation Software, Communication, Smart Meters
Landis+Gear	Automation Software, Communication, Smart Meters
Cisco	Automation Software, Communication
Itron	Communication, Smart Meters
Sensus	Communication, Smart Meters
Aclara	Smart Meters
Electro Industries	Smart Meters
General Electric (GE)	Smart Meters
Honeywell International Inc.	Demand Response

Table 5.1: Identified major smart grid manufacturers and their supported components [51]

Heterogeneous hardware architectures are used in firmware images, however, many ICSs are based on the ARM architecture [43, 131, 223]. Additionally, as reported in Figure 5.5, most of our collected IED firmware images are identified as targeting ARM architecture (82%), followed by PowerPC (9%). On the other hand, Linux is the most encountered OS in our *Firmware Database*, with a prevalence of 90% compared to others (e.g., Windows). Thus, the focus of this work is mainly on the ARM-Linux-based firmware images.

[8] https://www.greentechmedia.com/research/report/the-networked-grid-150-report-and-rankings-2013. Accessed on Dec 20, 2020.

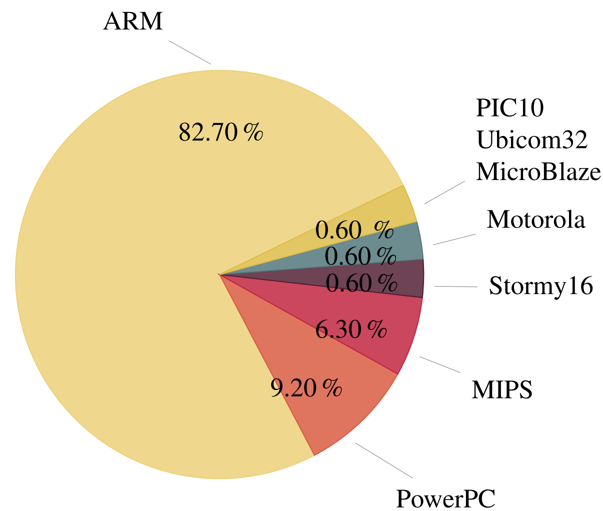


Figure 5.5: Distribution of hardware architectures amongst collected IED firmware

5.3.3 Vulnerability Database

Our study shows that many of the listed manufacturers reuse existing open-source libraries in their product implementations. This generally entails the legal obligation of publishing documents containing the licenses of all utilized open-source software. By investigating several sources of information pertaining to these manufacturers, such as corporate websites, product documentations and FTP search tools, we extract large amounts of open-source usage declarations that are related to the current smart grid scope. Some of the examples include distributed network protocol (DNP3), simple network management protocol (SNMP), network time protocol (NTP), and Sample Values (SV).

The top-25 relevant, popular and vulnerable open-source libraries are illustrated in Table 5.2, where they are ordered by their relative significance considering which ones are more frequently used in the IEDs of the identified manufacturers. Their significance is determined by multiplying the number of CVEs for a given library times the number of manufacturers that utilize that library. We download the source code of reused libraries with different versions, and cross-compile each of them for the ARM architecture using *GCC* compiler with four optimization flags (`O0-O3`). We cross reference with the CVE

database[9] to identify and label vulnerable functions with their corresponding CVEs. It is worth mentioning that all the functions of each library are stored in our *Vulnerability Database* for open-source library function identification. Additionally, the vulnerable functions are labelled by their correspondingly identified CVEs. Our *Vulnerability Database* consists of $3,270,165$ functions, of which $5,103$ are marked as vulnerable. This results in a total of 235 unique vulnerabilities after discarding the duplicates that result due to the use of different compilers and optimization flags.

Library	#CVEs	Manufacturers	Library	#CVEs	Manufacturers
1. php	601	Cisco, Honeywell, Siemens	14. qemu	225	Cisco
2. imagemagick	402	Cisco, GE, Honeywell	15. libxml2	44	ABB, Cisco, GE, Honeywell, Siemens
3. openssl	189	ABB, Cisco, GE, Honeywell, SE, Siemens	16. bind	102	Cisco, Siemens
4. mysql	564	Cisco	17. binutils	97	Cisco, Siemens
5. tcpdump	162	Cisco, GE, Siemens	18. libcurl	34	ABB, Cisco, Honeywell, SE, Siemens
6. openssh	87	ABB, Cisco, GE, Honeywell, Siemens	19. freetype	83	Cisco, Siemens
7. ntp	79	Cisco, GE, Honeywell, SE, Siemens	20. libpng	47	Cisco, Honeywell, Siemens
8. libtiff	149	Cisco, GE	21. samba	124	Honeywell
9. postgresql	98	Cisco, Honeywell, Siemens	22. utillinux	15	ABB, Cisco, GE, Honeywell, SE, Siemens
10. ffmpeg	274	Siemens	23. cups	88	Cisco
11. pcre	49	ABB, Cisco, GE, Honeywell, Siemens	24. lighttpd	28	ABB, Cisco, Honeywell
12. python	81	Cisco, Honeywell, Siemens	25. netsnmp	21	Cisco, GE, SE, Siemens
13. glibc	81	Cisco, Honeywell, Siemens			

Note: (GE): General Electric, (SE): Schneider Electric.

Table 5.2: Top-25 vulnerable open-source libraries in identified manufacturers [51]

The acquired firmware images contain various kinds of binaries, such as open-source, application-level, kernel and proprietary libraries. Consequently, by utilizing the CVE database, we identified $4,344$ CVEs in kernel-level, along with $5,581$ CVEs in application-level, and $2,336$ CVEs in open-source libraries amongst the identified manufacturers (considering the fact that some of the open-source libraries are reused in applications). Additionally, we have prepared an initial list of IED-specific proprietary libraries (e.g., NI). However, our list of such proprietary libraries is not yet comprehensive. Further effort would also be required in order to verify the identified vulnerabilities, since the source code of such proprietary libraries is not publicly available.

[9] https://github.com/cve-search/cve-search. Accessed on Dec 20, 2020.

5.3.4 Firmware Database

The proposed methodology is not necessarily specific to smart grid IEDs and therefore could be applied to any ARM-based binary code, such as IoT devices, routers, and IEDs. However, since the goal of this work is to assess the security of IEDs in the smart grid, we focus on the IED-specific firmware. Firmware reverse engineering is a time consuming and challenging task and requires domain expertise. In the following we provide more information about the firmware acquisition and firmware analysis challenges.

Firmware acquisition.

We first utilize popular FTP search engines, such as *NAPALM indexer*[10], *FileSearching*[11], and *fileWatcher*[12] to leverage publicly accessible corporate FTP servers. We then create a simple website scraper using *Scrapy*[13] and apply it to specific parts of each manufacturers' website. Finally, we perform a manual inspection for dynamically generated websites, which mostly applies to each manufacturers' download centre. All retrieved images are filtered based on the relevance to the smart grid context, and $2,628$ firmware packages are extracted. It is worth noting that sometimes the vendors do not provide firmware images in order to protect their IP or limit the access to it. Therefore, it might be necessary to directly extract or dump it from a device chip memory in different ways, such as an EEPROM programmer, bus monitoring during code upload and schematic extraction [203].

Firmware Analysis Challenges.

Performing firmware analysis with the objective of complete disassembly is challenging [54] and encounters additional challenges beside traditional binary analysis challenges presented in Chapter 2. This is partially due to a large requirement of time, domain specific

[10] https://www.searchftps.net. Accessed on Dec 20, 2020.
[11] http://www.filesearching.com. Accessed on Dec 20, 2020.
[12] https://2600index.info/Links/27/3/www.filewatcher.com. Accessed on Dec 20, 2020.
[13] https://scrapy.org. Accessed on Dec 20, 2020.

knowledge and research [130]. Furthermore, binaries are often stored in proprietary formats, obfuscated or encrypted for protection. These processes effectively make it extremely difficult (e.g., obfuscation [130]), or even impossible (e.g., uncrackable encryption [195], indecipherable formats [143]) to directly access the contents of a given binary blob. Encrypted binaries can sometimes be identified by their use of specific headers. For instance, a file encrypted with `OpenSSL` starts with the first 8-byte signature of "`Salted__`". In order to process all acquired firmware, we follow well-known procedures such as the ones presented in [195, 218]. This process has several main steps:

(i) *Unpacking and extraction*: Some vendors pack their firmware using proprietary packers and file formats, or use private key encryption. In practice, different unpacking tools, such as BINWALK[14], BAT [103], and FRAK [57] can be utilized to extract the firmware. However, performing such tasks cannot be always performed successfully, and thus not all firmware images can be analyzed.

(ii) *Firmware and binary identification*: Once the firmware is extracted, filtering is required for obtaining all relevant information. This can include binary files, configuration files, embedded files and the firmware itself. To this end, file signature matching is performed using different tools, such as SIGNSRCH[15], FILEFILE[16], and BINWALK. There exist some types of firmware that have no underlying operating system. They consist of only one binary file that operates directly on the hardware. In some cases, there is no abstraction of the OS and libraries, and in other cases, firmware images are not standard and no documentation is provided. Therefore, initializing a run-time environment and loading the binary is more challenging [192].

[14] `https://github.com/devttys0/binwalk`. Accessed on Dec 20, 2020.
[15] `http://aluigi.altervista.org/mytoolz.htm`. Accessed on Dec 20, 2020.
[16] `https://linux.die.net/man/1/file`. Accessed on Dec 20, 2020.

(iii) *Binary disassembling*: Finaly the firmware image should be disassembled by utilizing a disassembler (e.g., IDA PRO[17]), where using the properly identified architecture and entry point is required. The hardware architecture identification can be performed by the disassemblers or other powerful alternatives such as BINWALK. As for the entry point, a given binary blob can contain several entry points [192], and it may not be possible for tools such as IDA PRO to automatically identify them. In these cases entry point discovery should be performed [130, 192], which is one of the most challenging parts of this entire procedure and requires leveraging various techniques (e.g., [223]).

5.4 Multi-stage Detection Engine

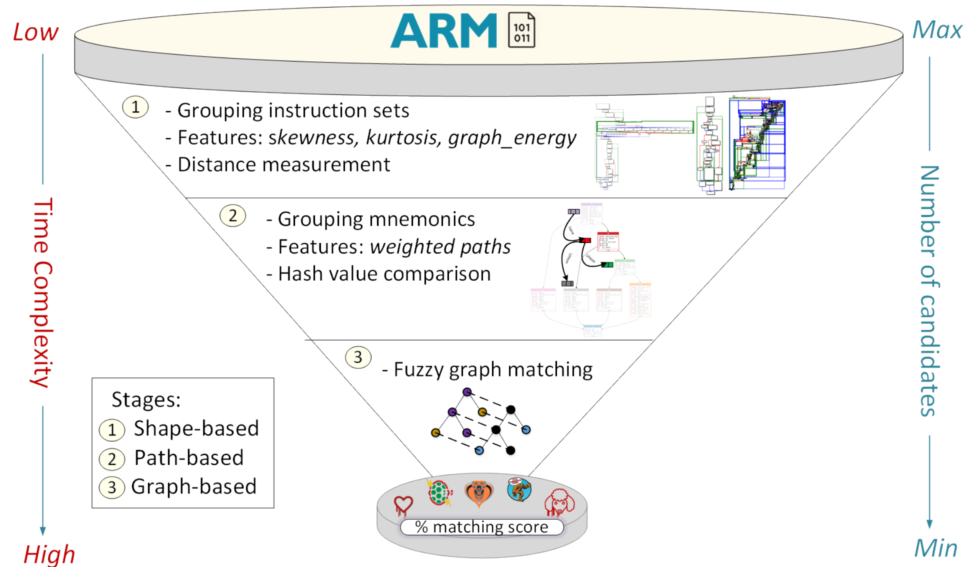


Figure 5.6: BINARM: multi-stage detection engine

We propose an efficient multi-stage detection engine as shown in Figure 5.6 to identify vulnerable functions in firmware images, which involves three detection stages, from coarse to

[17] https://www.hex-rays.com/products/ida/. Accessed on Dec 20, 2020.

granular. Our key idea is to start with light-weight feature extraction and function matching operations, and to perform the most expensive operations only for a reduced set of candidates. More specifically:

(i) Function shape-based detector extracts the simplest and more distinguishable features that quickly eliminates dissimilar candidates with less computational overhead.

(ii) Path-based detector performs more expensive matching operations, however, still not as expensive as graph matching. Specifically, it extracts execution paths including their corresponding instruction sequences, then turns them into hash values, and simply employs a binary search.

(iii) Fuzzy matching graph-based detector performs the most expensive operations, which mainly includes careful examination of basic blocks, their neighbours, and graph matching for a selected and relatively smaller number of candidates.

The details of each stage are explained in the following.

5.4.1 Function Shape-Based Detection

The function shape-based detection is performed using a collection of heterogeneous features extracted at different levels of a function, namely, function *shape* [190], as introduced in Chapter 4. This includes *instruction-level* features, *structural* features, and *statistical* features. Therefore, we extract various features from all the functions in our *Vulnerability* database. The corresponding details are briefly described in the following.

Feature Extraction

The *instruction-level* features carry the syntax and semantic information of a function [11], such as the number of caller functions (`#callers`). For instance, the frequencies of strings have been used to classify malware based on their behaviour [182]. To capture

Instruction-level	Structural	Statistical
retType	rich_club_metric	mean
#instructions	#nodes	variance
localvarsize	average_degree	skewness
#arguments	#edges	kurtosis
#strings	cyclomatic complexity	Z-score
#mnemonics	average_path_length	standard deviation
xrefs	graph_energy	
#constants	link_density	
#registers	algebraic_connectivity	
#operands	height of the root	
refnames	s_metric	
arguments	pearson	
flags	num_conn_triples	
operands	leaf_nodes	
declaration		
argsize		

Table 5.3: Function shape features

the topology of a function and to extract the *structural* features, we employ a set of graph metrics [96], such as cyclomatic complexity, and `s_metric`. However, some functions might have the same structural shape, while being semantically different. As a result, we consider additional features in order to include more semantic information. Finally, *statistical features* are used in order to capture the semantics of a function [178], such as `skewness` and `kurtosis` [170], which are extracted as follows:

$$S_k = \left(\frac{\sqrt{N(N-1)}}{N-1}\right)\left(\frac{\sum_{i=1}^{N}(Y_i - \overline{Y})^3/N}{s^3}\right)$$

$$K_z = \frac{\sum_{i=1}^{N}(Y_i - \overline{Y})^4/N}{s^4} - 3$$

where N is the number of data points, Y_i is the frequency of each instruction, $\overline{Y}$ represents the *mean*, and s is *standard deviation*. An excerpt of the extracted features is listed in Table 5.3. We refer the reader to Chapter 4 for the complete list of the features.

Normalization

In the ARM instruction set, each assembly instruction consists of a mnemonic and a sequence of up to five operands. Two fragments of code might be identical both structurally and syntactically, but different in terms of memory references or registers. Hence, it is essential to normalize the instruction sets prior to comparison. For this purpose, we normalize the operands according to the mapping sets provided by IDA PRO as listed in Table 5.4. We further categorize the "general" registers based on their types[18], as presented in Table 5.5. For instance, the `MOV R3,R3,ASR#8` instruction has three operands and will be normalized to `MOV 100100600`, while the `STR R0,[R11,#ctx]` instruction will be normalized to `STR 100300`.

IDA Operand Type	IDA Operand Mapping	BINARM Normalization
None	0	000
General Register	1	1** (Refer to Table 5.5)
Memory Reference	2	200
Base + Index	3	300
Base + Index + Displacement	4	300
Immediate	5	400
Immediate Far Address	6	500
Immediate Near Address	7	500
FPP Register	8	600
386 Control Register	9	700
386 Debug Register	10	700
386 Trace Register	11	700
Condition (for Z80)	12	800
bit (8051)	13	900
bitnot (8051)	14	900
Other	Else	NULL

Table 5.4: Instruction normalization

[18] http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0068b/CIHEDHIF.html. Accessed on Dec 20, 2020.

Feature Selection

After collecting and extracting all heterogeneous features from our *Vulnerability Database*, we aim at identifying the features that differentiate different functions the most in order to improve the accuracy and efficiency of the shape-based function detection. To this end, mutual information (MI) [173] is leveraged to measure the dependency degree between the aforementioned features and the functions in *Vulnerability Database*. The mutual information is implemented in *python* and *Science Kit Learn* machine learning library [137, 172]. The obtained results of the mutual information process are in the form of information bits (the higher values give more information).

Based on the obtained results from the mutual information shown in Figure 5.7, we choose three top-ranked features, `graph_energy`, `skewness` (`sk`), and `kurtosis` (`kz`), as a 3-tuple feature for each function. It is worth mentioning that there is a dependency between the next top-ranked features and `graph_energy`. However, based on our experiments extracting the five top-ranked features instead of the top three would not have

Register Name	Partial Normalization	Concerned Registers	BINARM Norm.
General	10*	a1-a4, r0-r3	100
		v1-v5 , r4-r8	101
		v6, sb, SB, r9	102
		v7, sl, SL, r10	103
		v8, fp, FP, r11	104
		ip, IP, r12	105
		sp, SP, r13	106
		lr, LR, r14	107
		pc, PC, r15	108
Program Status	11*	CPSR	110
		SPSR	111
Floating Point	12*	f0-f7 , F0-F7	121
		s0-s31 , S0-S31	122
		d0-d15 , D0-D15	123
Co-processor	13*	p0-p15	130
		c0-c150	131
Other	14*	NA	140

Table 5.5: General register normalization

made a significant difference in the results of the filtering process according to our dataset and threshold. Additionally, since our goal is to perform coarse detection at this stage, and extracting more features would affect the time complexity, we choose the first three top-ranked features. Our experiments confirm the effectiveness of the three chosen features (shown in Section 5.5.6).

Function Matching

To perform a coarse detection process for a given target function against a set of reference functions from our repository, we first utilize a function shape-based detection strategy.

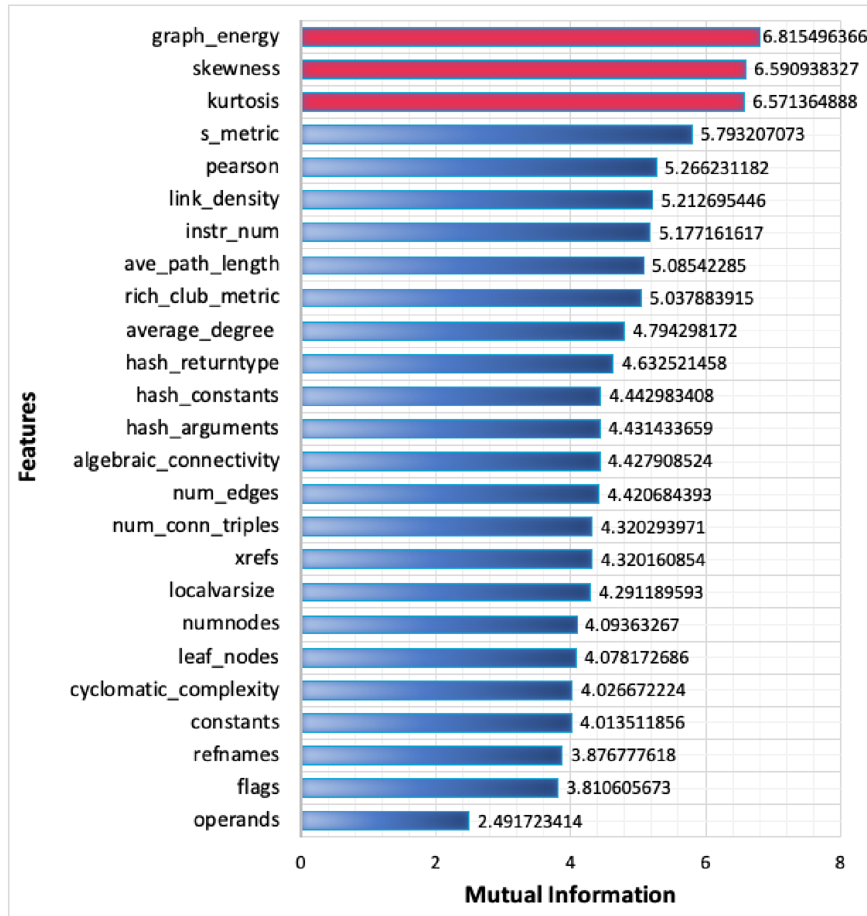


Figure 5.7: Mutual information of features

Therefore, a similarity metrics should be defined. Consequently, all functions in our *Vulnerability Database* that surpass a predetermined threshold distance, λ, from a given target function are deemed dissimilar in shape-based detection stage. Euclidean distance is used to calculate the distance between two given functions as follows:

$$dist(p, q) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}$$

where $p = (p_1, p_2, p_3)$ and $q = (q_1, q_2, q_3)$ are two points in Euclidean 3-space, representing the `graph_energy`, `skewness`, and `kurtosis` features for the target and reference functions, respectively. In order to calculate the threshold distance, we employ K-Means clustering [100, 115] (Section 5.4.1) on the obtained features and based on the distances in the clusters, the final threshold distance $\lambda = 26.45$ is calculated. Therefore, the functions having points further apart than the threshold value of $\lambda = 26.45$ are deemed dissimilar.

Definition 2 *Let f_T, f_r be two functions, and p and q be 3-tuple associated with f_T, f_r. Let $p \leftarrow GSK(f_T)$ and $q \leftarrow GSK(f_r)$ extract* `graph_energy`*,* `skewness`*, and* `kurtosis` *features from f_T and f_r functions, respectively. Let $dist(p, q)$ be a Euclidean distance function ($dist \geqslant 0$) and λ a predefined threshold value ($\lambda > 0$). We consider f_r as a candidate function to be matched against f_T, if $dist(p, q) \leqslant \lambda$.*

Threshold Selection

As mentioned in the previous subsection, the functions that have a distance less than λ from the target function are considered as potential candidates. On the other hand, obtaining such a threshold value is challenging and one possible solution could be to experimentally derive it. However, we obtain the threshold value automatically and empirically by leveraging K-Means clustering algorithm, which groups similar functions associated with the 3-tuple in the same cluster in an unsupervised manner. K-Means clustering algorithm partitions n

observations into k clusters, $C_1, \ldots, C_k$, such that the total within-cluster sum of square (WSS) [101] is minimized as follows:

$$WSS = \sum_{i=1}^{k} \sum_{p \in C_i} dist(p, c_i)^2 \tag{14}$$

where p represents a given observation; c_i is the centroid of cluster C_i, and $dist$ is the Euclidean distance. To identify the optimal number of clusters, we employ the elbow method [66], where the goal is to get a small WSS while minimizing k. To this end, the 3-tuple features are extracted from all the functions in our *Vulnerability Database*. Then, K-means clustering is applied to our data points for each value of k starting from one to 100, and the WSS is calculated. The optimal value for k is at the drop off point (knee shape) [51], which in our cases is equal to 11.

The ultimate goal of the clustering is to acquire a distance threshold value in order to drop some of the candidate functions which are far from the target function based on the 3-tuple features. In order to get the threshold value, first we calculate the average Euclidean distances of all 3-tuple points in each cluster to acquire how much further apart the similar functions are. Finally, the average of eleven obtained distances is calculated and considered as the final threshold value λ, which is equal to 26.45 for our *Vulnerability Database*. More formally,

Definition 3 *Let $\{f_1, f_2, \ldots, f_n\}$ be a set of functions in our repository. Let $p_i \leftarrow GSK(f_i)$ and $q_j \leftarrow GSK(f_j)$ generate a 3-tuple `graph_energy`, `skewness`, and `kurtosis` associated with f_i, and f_j functions, respectively, and let $dist(p_i, q_j)$ calculate the Euclidean distance between f_i and f_j functions. Let K and n_c denote the number of clusters and the number of points in each cluster, respectively. We calculate the threshold value $\lambda > 0$ as follows:*

$$\lambda = \frac{1}{K} \times \sum_{c=1}^{K} \left(\frac{\sum_{i=1,j=1}^{n_c} dist(p_i, q_j)}{n_c} \right)$$

5.4.2 Path-Based Detection

After detecting similar functions based on the Euclidean distances of their shapes, BinArm incorporates a path-based detection to reduce the graph comparison effort during the final detection stage. The idea behind the path-based detector is that similar functions have similar execution paths. Moreover,, analyzing the execution paths has been used to identify function vulnerabilities as well as stealthy program attacks [194, 208]. Accordingly, we propose a weighed path comparison algorithm to measure the similarity of two functions based on their branch executions.

Weighted Normalized Tree Distance (WNTD)

The normalized tree distance (NTD) [222] is proposed for comparing phylogenetic trees with the same topology and same set of N taxonomic groups including the lengths of the edges. Consider two phylogenetic trees A and B denoted by $A = \{a_1, a_2, \ldots, a_N\}$ and $B = \{b_1, b_2, \ldots, b_N\}$, where N is the number of paths and a_i and b_i are the lengths of path i in trees A and B, respectively. In order to compare trees A and B, the distance is measured as follows [222]:

$$NTD = \frac{1}{2}\left(\sum_{i=1}^{N}\left|\frac{a_i}{\sum_{j=1}^{N} a_j} - \frac{b_i}{\sum_{j=1}^{N} b_j}\right|\right)$$

Such a dissimilarity metric scales from 0 (identical trees) to 1 (distinct trees). However, NTD is originally designed for two trees with the same topology (the same number of paths). Additionally, NTD does not consider the contents of nodes and the paths will be directly compared without taking into account their content similarities.

Inspired by NTD, we propose a weighted normalized tree distance (WNTD) metric to measure dissimilarity between two functions W and V. First, we consider the CFGs of the two functions, represent them as a directed acyclic graph by unrolling the loops, and then

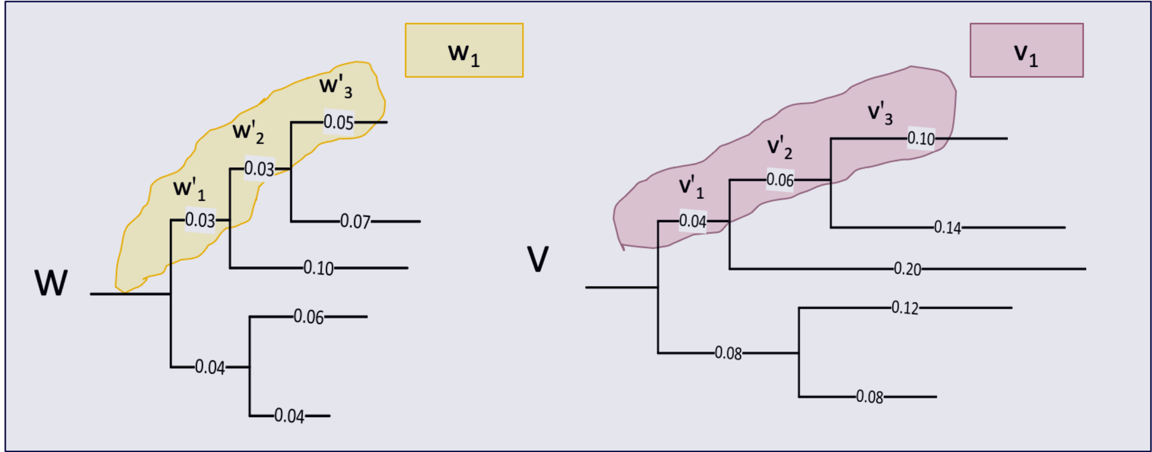


Figure 5.8: An example of two *weighted paths* of $w1$ and $v1$

extract all possible paths from the two CFGs using breadth first search (BFS). Based on the contents of basic blocks along each path and their neighbours, a corresponding weight (e.g., w_i and v_i) is calculated and assigned to each path of the two functions, which is named "*weighted paths*" (shown in Figure 5.8). Second, for any given weighted path i extracted from function W, we find the best match v_{BM} in function V and calculate the ratio and the final dissimilarity. More formally,

Definition 4 *Let* $W = \{w_1, w_2, \ldots, w_N\}$ *and* $V = \{v_1, v_2, \ldots, v_M\}$ *denote two functions containing* N *and respectively* M *(*$N \leq M$*) number of weights representative of their weighted paths. The dissimilarity between the functions* W *and* V *is measured as follows:*

$$WNTD = \frac{1}{2}\left(\sum_{i=1}^{N}\left|\frac{w_i}{\sum_{j=1}^{N}(wj)} - \frac{v_{BM}}{\sum_{j=1}^{M}(v_j)}\right|\right) \tag{15}$$

where w_i *and* v_i *are the weighted paths of functions* W *and* V*, respectively; and* v_{BM} *is the best match for weighted path* w_i *amongst the other weighted paths in function* V*.*

WNTD considers a weight for each basic block and finally a single weight for each path of a function. Moreover, even if the two CFGs do not have the same number of paths, still the two CFGs can be compared (for every path, a match can be found as either the best

match or zero). Once the WNTD comparison is performed, the functions with a distance less than γ are preserved for the final detection step. Performed experiments (Section 5.5.7) suggest 0.5 cut off is the best.

Mnemonic Instructions Grouping

Instruction mnemonics carry information about the semantics of a function, for instance, cryptographic functions perform more logical and mathematical operations compared to a function which opens a file. For instance, opcode frequencies are used to detect metamorphic malware [178, 202]. Hence, extracting the frequencies of different kinds of instructions can provide some clues about the functionality of a function. However, due to

Group Name	Examples
Branch	B, BL, BX, BLX, BXJ, BNE, IT, CBZ, CBNZ, TBB, TBH
Arithmatic	ADD, ADC, SUB, SBC, RSB, RSC, SDIV, UDIV, USAD8, USADA8
Logical	AND, EOR, ORR, BIC, EORS
Saturating	QADD, QSUB, QDADD, QDSUB, SSAT, USAT, SSAT16, USAT16
BarrelShifter	LSL, LSR, ASR, ROR,RRX
Multiplication	MUL ,MLA, MLS, MULL, MLAL, UMULL, UMLAL, SMULL, SMLAL, SMULxy, SMLAxy, SMULWy, SMLAWy, SMLALxy, SMUAD, SMUSD, SMMUL, SMMLA, SMMLS, SMLAD, SMLSD, SMLALD, MLSLD, UMAAL, MIA, MIAPH, MIAxy
ReverseBytes	REV, REV16, REVSH, RBIT
Comparison	CMP, CMN, TST, TEQ
DataMovement	MOV, MVN, MOVS, MOVT
LoadStore	LDR, STR, ADR , LDRB, STRB, LDRH, STRH, LDRSB,LDRSH, LDREQB, PLD, PLDW, PLI, LDM, STM, LDREX, STREX, ADRL, MOV32, UND
Transfer	MRS,MSR
Stack	PUSH, POP, STMFD, LDMFD, STMFA, LDMFA, STMED, LDMED, STMEA, LDMEA, STMIA, LDMIA, STMIB, LDMIB, STMDA, LDMDA, STMDB, LDMDB
Swap	SWP, SWPB
PackUnpack	BFC, BFI, SBFX, UBFX, SXT, SXTA, UXT, UXTA, PKHBT, PKHTB
CoProcDataProcessing	CPD, CDP2
CoProcRegTransfer	MRC, MCR, MCR2, MCRR, MCRR2, MRC2, MRRC, MRRC2
CoProcMemTransfer	LDC, STC, LDC2, STC2

Table 5.6: Proposed menmonic groups for ARM instruction set

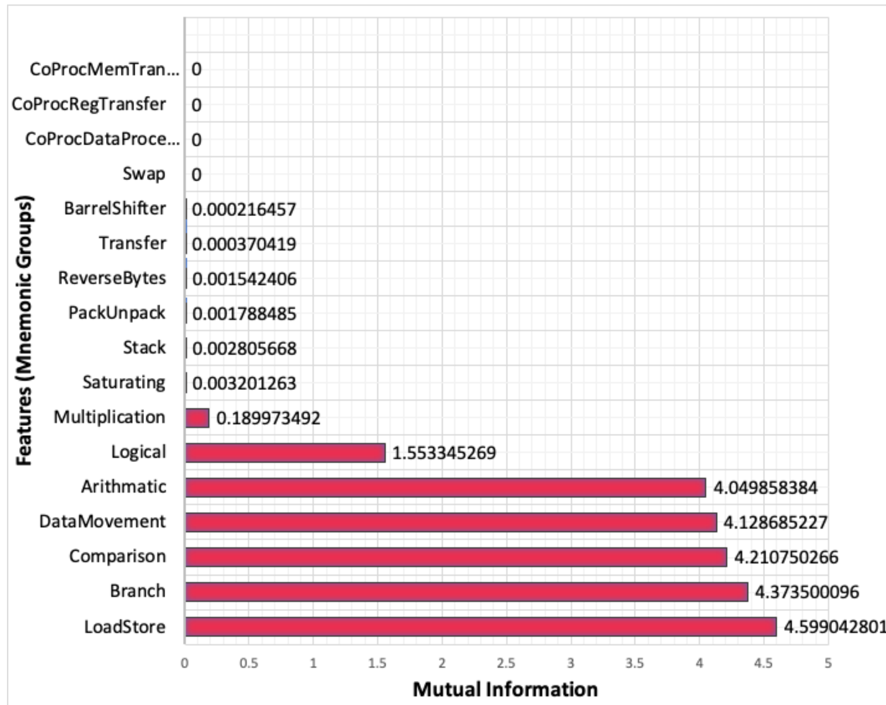


Figure 5.9: Mutual information of mnemonics

various factors, such as compiler effects, different mnemonics might be used interchangeably. Therefore, we identify the list of ARM instruction set mnemonics, and group them based on their functionalities, e.g., arithmetic instructions. We obtain seventeen groups of mnemonics which is listed in Table 5.6. For instance, any instruction mnemonic that is part of {ADD, ADC, SUB, SBC, RSB, RSC, SDIV, UDIV, USAD8, USADA8} list, will be grouped into Arithmatic category.

We further intend to identify the most relevant and informative mnemonics to distinguish amongst functions. As such, we leverage mutual information (MI) to measure the dependency degree between mnemonic group frequencies and functions in our *Vulnerability dataset*. Obtained results are presented in Figure 5.9. Accordingly, we choose the 7-top-ranked mnemonic groups as the features to be extracted from each basic block in a path and utilize them to compute the weighted paths.

Weight Assignments

In order to calculate the WNTD, the *weighted paths* need to be calculated. To condense all the information of a node and its neighbours into a single hash value, a graph kernel with linear time complexity is proposed in [93, 104]. Inspired by this approach, we create a single hash value to be representative of each *weighted path*. Therefore, we calculate the accumulated weights of each node along the path and assign a single hash value to each path to compute the *weighted paths*. In order to consider more information, the weight assigned to each node is calculated based on the top-ranked instruction groups of the node itself and also its neighbours (parents and children) that could be out of the current path.

For this purpose, for each basic block and its immediate neighbours in a given path, we first extract top-ranked mnemonic groups and create a feature vector of their probability density functions (PDF)[19] per basic block. We further distinguish between the in-degrees (parents) and out-degrees (children) to bring more information about the execution flow. To this end, we calculate the joint and the union of the PDFs for the parents and children, respectively. This results into a feature vector of PDFs for each node. An example is illustrated in Figure 5.10, where the chosen path contains the nodes with IDs 0, 2, 4 and 6 and the PDF feature vectors that are used to generate the final PDF of node 2 are highlighted. Finally, the Trend Micro locality sensitive hash (TLSH)[20] [171] is applied on the obtained feature vectors and a weight is assigned to each node in a given path. The final *weighted path* is computed by the summation of all hash values associated to each node along the path. The details of the approach are presented in Algorithm 3.

Finding the Best Match

After calculating all the weighted paths for both functions (e.g., w_i and v_i), we need to find the best matching pairs. In order to find the best match (e.g., v_{BM} in Equation 15) for each path, e.g., w_i, we pre-calculate all the weights of all paths for both reference and target functions foremost, and store the obtained weighted paths of the larger function V in a *B^+tree*. Afterwards, we perform exact and

[19] https://wiki.ubc.ca/Science:MATH105_Probability/Lesson_1_DRV/1.03_The_Discrete_PDF. Accessed on Dec 20, 2020.

[20] https://github.com/trendmicro/tlsh. Accessed on Dec 20, 2020.

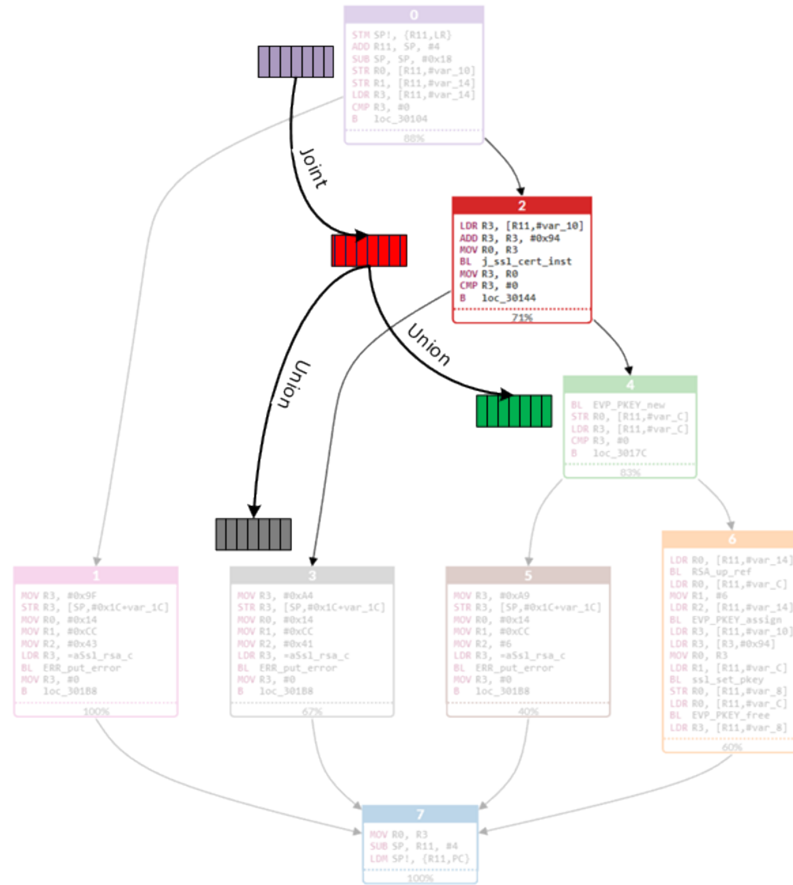


Figure 5.10: Weight assignment example

inexact matching to determine the best match for weighted paths. More formally:

$$v_{BM} = \begin{cases} excatMatch(w_i, \vec{V}) & \text{, if there is any match} \\ inexactMatch(w_i, \vec{V}, \delta) & \text{, if there is any match} \leqslant \delta \\ 0 & \text{, else} \end{cases} \tag{16}$$

First, we search in the B^+*tree* to find the exact match for each weight in function W, and then remove it from the B^+*tree*. Second, we perform inexact matching by considering *backward* and *forward* sibling pointers to each leaf node [95], which points to the previous and next leaf nodes, respectively. The number of neighbours is obtained by a user-defined distance δ. If there is no match for a given path, the best match would be zero. Due to the usage of B^+*tree*, the time complexity to find the best match is $O(n \log m)$, where n and m are the number of weighted paths in functions W

Algorithm 3: Weight assignment

```
   Input: Path_a : A path extracted from the CFG.
   Output: w : Weighted path.
   Initialization
1  f[] ← []; // PDF of top-ranked instruction groups
2  weights[] ← []; // Feature vector of the weights
3  w ← 0; // Initialize the path weight to zero
4  begin
       foreach node[i] ∈ Path_a do
5          f ← node[i].getPDF();
6          U[] ← [];
7          J[] ← f;
8          while (node[i].hasParents()) do
9              J ← J ∩ node[i][].getParent().getPDF();
10         end
11         while (node[i].hasChildren()) do
12             U ← U ∪ node[i][].getChild().getPDF();
13         end
14         f ← J + U;
15         weights[i] ← TLSH(f);
16     end
17     foreach wt[i] ∈ weights do
18         w ← w + wt[i];
19     end
20     return w;
   end
```

and V. Finally, the detailed procedure of calculating the WNTD is presented in Algorithm 4.

5.4.3 Fuzzy Matching-Based Detection

The results of the path-based detection stage, which correspond to a relatively small set of candidate functions, are passed to the final detection stage. In order to compare a given target function to the reference functions in the candidate set, inspired by [110], we perform fuzzy matching on each pair of functions and obtain the similarity score. Functions with the highest similarity scores are returned as the final matching pairs. The details are described in the following.

Algorithm 4: WNTD

Input: W: Weighted paths of function W stored in a linked list.
Input: $BTree_V$: Weighted paths of function V stored in a B$^+$tree.
Output: $WNTD$: Dissimilarity score between functions W and V.

```
1  Function WNTD(W, BTree_V)
2      sum ← 0 ;
3      sum_W ← Σ_{j=1}^{N} (w[j]);
4      sum_V ← Σ_{j=1}^{M} (v[j]);
5      foreach w[i] ∈ W do
6          v_BM = exactMatch(BTree_V, w_i) ;
7          if v_BM ≠ −1 then
8              sum+ = | w[i]/sum_W − v_BM/sum_V |;
9              W.remove(w[i]);
10         end
11     end
12     v_BM ← 0 ;
13     foreach w[i] ∈ W do
14         v_BM = inexactMatch(BTree_V, w_i, δ) ;
15         sum+ = | w[i]/sum_W − v_BM/sum_V |
16     end
17     WNTD = sum/2;
18     return WNTD;
19 end
```

Path and Neighbourhood Exploration

The fuzzy matching approach is composed of three main phases: (i) longest path extraction; (ii) path exploration; and (iii) neighbourhood exploration, which is illustrated with an example in Figure 5.11. First, we unroll all the loops and employ depth first search on the CFG of the target function to extract the longest path (as depicted in Figure 5.11 part a). A path represents one complete particular execution, where its functionality is the result of executing all its basic blocks. Therefore, retrieving two equivalent paths can be an initiation point to further explore the immediate neighbours of their nodes. The longer the path is, the more matching pairs would be acquired.

Second, the reference function is explored to find the best match for the longest path in the target function. Inspired by [110, 145], a breadth-first search combined with longest common subsequence (LCS) method of dynamic programming [53] is performed. In order to satisfy the requirements of

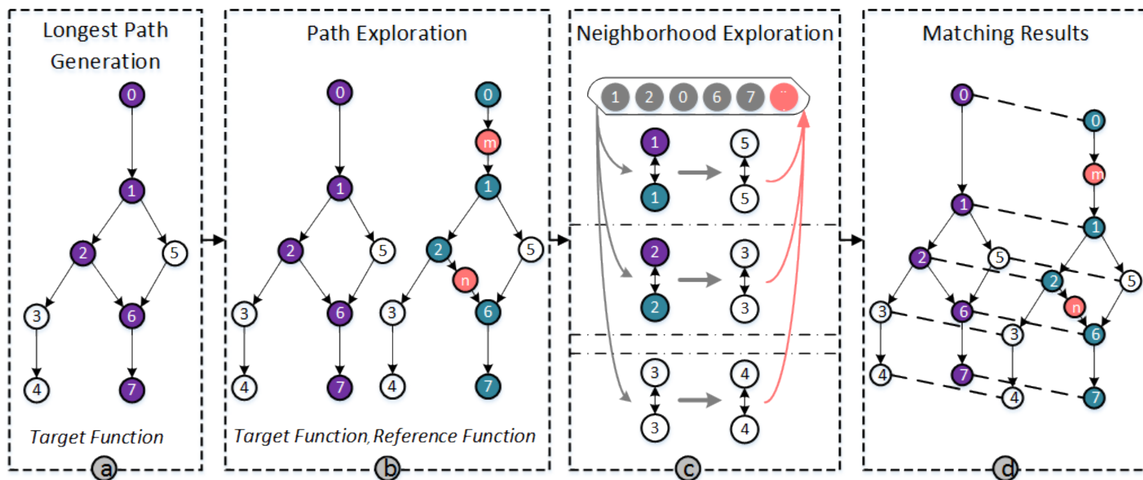


Figure 5.11: Fuzzy matching

the LCS algorithm, since any path is a sequence of basic blocks, each basic block is treated as a letter. Two basic blocks are compared based on their instructions, and a similarity score (Subsection 5.4.3) is returned. Therefore, all the possible paths in the reference function are explored and the one with the highest similarity score is returned as the best matched path (including basic blocks pairs) [110]. As an example, the best match for the given longest path with a reference function is highlighted in Figure 5.11 part b. Besides, we put all the obtained matching basic blocks pairs in a priority queue.

Finally, we perform neighbourhood exploration and leverage Hungarian algorithm in both target and reference functions to improve and extend the mapping. Since all the mapping basic block pairs are obtained during the path exploration process, we start by exploring the neighbours of the most similar basic block pairs (priority queue shown in Figure 5.11 part c). Therefore, we initiate the search by the most similar basic blocks and find more matched pairs for their successors and predecessors by considering their in-degrees and out-degrees and leveraging Hungarian algorithm. In the case of a new match, we put the matched pairs in the priority queue to explore their neighbours later on. We continue the same algorithm for the rest of the nodes until the priority queue is empty.

In our example, consider the obtained priority queue from path explorations as presented on top of the box in Figure 5.11 part c. The highest obtained similarity score is related to basic blocks with IDs 1 in both target and reference functions; therefore, we initiate the search from there. Since both nodes have the same in-degree and out-degree numbers, we check the number of their successors.

Whereof both nodes have more than one successor, the Hungarian algorithm is leveraged to find the best mappings between the two sets of successors, otherwise their successors would be matched directly. However, if we have already paired a node with another one, the corresponding match will be discarded. Therefore, since the nodes with ID 2 are already matched, we discard them and consider the second successor (5) as a match and put in the priority queue to explore its neighbours later on. Further, we check the predecessors of the nodes with ID 1. Both have one predecessor, while the predecessor of the node in target function is already matched, therefore it will be discarded. We remove node 1 from the priority queue and continue the same algorithm for the rest of the nodes until the priority queue is empty. The final matching graphs are depicted in Figure 5.11 part d.

The outcome of neighbourhood exploration is basic block matching pairs in a CFG (Figure 5.11 part d) and the corresponding similarity scores. We measure the final similarity score between f_T and f_r functions having n_T and n_r number of basic blocks, respectively, as the following:

$$similarity\ (f_T, f_r) = \frac{2 \times \sum_{i=1}^{k} WJ(S,T)}{n_T + n_r} \tag{17}$$

where k is the number of matched basic blocks between functions f_T and f_r, and $WJ(S,T)$ returns the similarity score between the matching basic block pairs. Therefore, BINARM provides all the differences between two functions at instruction level, basic blocks level and function level.

Basic Block Matching

For basic block matching, we could adopt the LCS method of dynamic programming on the instructions of two basic blocks as in [110]. However, the accuracy of this approach might be affected by *instruction reordering* and *instruction substitutions* [110]. Moreover, the time complexity of the LCS algorithm is $O(mn)$, where m and n represent the number of instructions in the two basic blocks. Consequently, to accurately and efficiently perform basic block matching, we use the weighted Jaccard similarity [112] between the two basic blocks. Let S and T be two sets containing the mnemonic frequencies of the two basic blocks, with n and m number of elements in each of

thier blocks. The weighted Jaccard similarity (WJ) between the two vectors is calculated as follows:

$$WJ(S,T) = \frac{\sum_{k=1}^{N} \min(S_k \cap T_k)}{\sum_{k=1}^{N} \max(S_k \cup T_k)},\ N = \max\{m,n\}$$

The usage of WJ similarity together with instruction grouping could overcome *instruction reordering* and in some cases *instruction substitutions*. Moreover, the time complexity of the WJ similarity is of order $O(N)$.

5.5 Evaluation

This section outlines all our experimental work and analysis. We first elaborate on the environment setup. Then, we discuss the function identification accuracy of our tool, and then examine its efficiency and scalability. Moreover, we compare BINARM with state-of-the-art approaches. We further present a case study that shows how BINARM would function in real-world scenarios. Finally, we investigate and scrutinize the impact of our multi-stage detection approach and the selected parameters.

5.5.1 Experimental Setup

All of our experiments are conducted on machines running Windows 7 and Ubuntu 15.04 with Intel Xenon $E5$ 2.4 GHz CPU and $16GB$ RAM. BINARM is written in C++ and utilizes a *Cassandra* database[21] to store all the functions along with their features. A custom python script is used in tandem with IDA PRO to extract function CFGs in the desired JSON format. *Vagrant* is used to create a specialized environment used for firmware reverse engineering as well as library cross compilation for the *ARM* architecture. The utilized cross compiler is *gcc-arm-linux-gnueabi* version 4.7.3 using the debug flag, the static flag, and all compatible optimization flags (`O0-O3`). The symbol names are preserved during the compilation process for metric validation. *Docker*[22] is used

[21] http://cassandra.apache.org/. Accessed on Dec 20, 2020.
[22] https://www.docker.com/. Accessed on Dec 20, 2020.

to create a containerized version of the CVE database and its associated search tools[23]. *Python* is used in conjunction with the containerized CVE database to extract relevant CVE information related to identified libraries. The representational state transfer (REST) interface is utilized to perform function indexing and function matching.

Dataset. The experiments are performed on three different datasets: *Vulnerability* dataset, *Firmware* dataset and *General* dataset, which are explicitly indicated in each section. In order to evaluate the scalability of BINARM, a large number of IED and non-IED firmware images are collected from the wild, of which $5,756$ were successfully disassembled to construct our *General* dataset.

Evaluation Metrics. To evaluate the accuracy of BINARM, since our data is not imbalanced and we do not perform a direct accuracy comparison with state-of-the-art approaches, we use the total accuracy metric:

$$TA = \frac{TP + TN}{TP + TN + FP + FN}$$

where TP is the number of relevant functions retrieved correctly; FP represents the number of irrelevant functions that are incorrectly detected; and FN indicates the number of relevant functions that are not detected, and TN is the number of not-detected irrelevant functions.

Time Measurement. The execution time for function indexing is measured by adding the time required for each step, including feature extraction and function indexing. The search time includes time required for feature extraction and function discovery. The time taken to disassemble the binaries using IDA PRO is excluded, where it takes on the order of seconds on average to disassemble a binary file and can be distributed over all functions in a binary file.

5.5.2 Library Function Identification Accuracy

We evaluate the accuracy of BINARM by examining a randomly selected set of open-source libraries from our *Vulnerability Database*, where the source code and the symbol names are provided in order to validate the results. We randomly select 10% of libraries from *Vulnerability Database* as targets, and match them against the remaining 90% of libraries in our repository. We repeat this process

[23] https://github.com/leojcollard/cve-search-docker. Accessed on Dec 20, 2020.

various times. The average accuracy results are summarized in Table 5.7. As can be seen, the average of total accuracy is 0.92. According to our experiments, the results can vary due to different versions and the degree of changes in the new versions. Since the libraries are randomly selected, in some cases the differences between versions are relatively high, resulting in a drop in the accuracy.

Library	**Accuracy**
glibc	0.96
libcurl	0.93
libxml2	0.89
lighttpd	0.92
ntp	0.87
openssh	0.89
openssl	0.93
postgresql	0.98
zlib	0.89
Average	**0.92**

Table 5.7: Accuracy results for library function detection

5.5.3 Efficiency

In this section, we detail the conducted experiments to measure the efficiency of BINARM for function matching. To this end, we test the $5,103$ vulnerable functions against all functions in our *Vulnerability Database* and *Netgear ReadyNAS v6.1.6*[24] firmware separately, and measure the search time for each function.

The obtained results are reported in Figure 5.12, where the x-axis represents the percentage of number of functions, and the y-axis shows the cumulative distribution function (CDFs) of search time. The searching time average values per function for each scenario are 0.01 seconds and 0.008 seconds, respectively. Note that the search time of BINARM is strongly related to the CFG complexity of the target function. If the target function has a large value of graph_energy, the search time would be higher. However, the search time of a small function against a very complex CFG

[24] http://www.downloads.netgear.com/files/GDC/READYNAS-100/ReadyNASOS-6.1.6-arm.zip. Accessed on Dec 20, 2020.

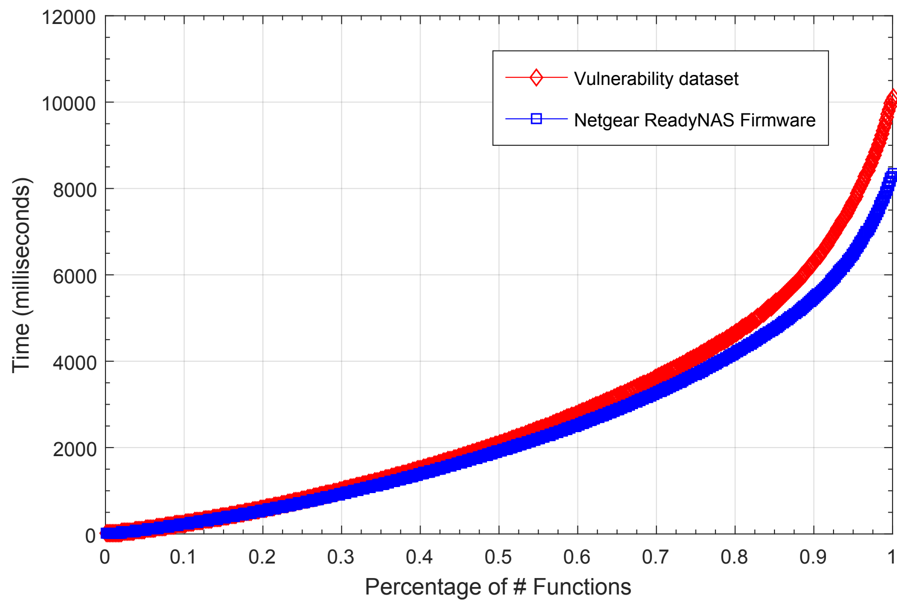


Figure 5.12: CDF of vulnerable function search time

would not be costly, since the complex functions are deemed dissimilar in the shape-based detection stage and filtered out, and no heavy graph matching would be performed.

5.5.4 Comparison

In this section, we compare BINARM with the state-of-the-art approaches.

Indexing Time Comparison

In order to compare the indexing time of BINARM with the state-of-the-art DISCOVRE [78], GENIUS [84], and MULTI-MH [174] approaches, we choose the *Netgear ReadyNAS v6.1.6*[25] firmware image. The reasons of this choice are threefold: (i) the firmware is publicly available and is based on the ARM architecture; (ii) all the aforesaid works have measured the indexing time of Netgear ReadyNAS based on their techniques; and (iii) the hardware specifications of the machines used for the experiments are provided. Altogether these facilitate the comparison.

[25] http://www.downloads.netgear.com/files/GDC/READYNAS-100/ReadyNASOS-6.1.6-arm.zip. Accessed on Dec 20, 2020.

	MULTI-MH [174]	GENIUS [84]	BINARM	DISCOVRE [78]
Time	5, 475	89.7	78.65	54.1
Hardware Specification	Intel Core i7-2640M at 2.8GHz 8GB DDR3-RAM	24 Cores at 2.8 GHz 65GB RAM	Intel Xenon E5-2630v3 at 2.4 GHz 16GB RAM	Intel Core i7-2720QM at 2.20 GHz 8GB DDR3 RAM

Table 5.8: Baseline comparison on indexing time (in minutes) of `ReadyNAS v6.1.6`

We index *ReadyNAS* in our database and record the indexing time. Table 5.8 illustrates the preparation time along with the hardware specifications that are reported by the aforementioned approaches, as well as those of BINARM. By taking the machines computational power into account, BINARM is more efficient with respect to indexing time when compared to aforesaid approaches with the exception of DISCOVRE. The reason is that DISCOVRE only considers CFG extraction time, while BINARM extracts additional features, such as the weighted paths. Nevertheless, the evaluation performed by [84] demonstrate DISCOVRE's inaccuracy in large scale setup.

Search Time Comparison

We further compare the search time of our prototype system with that of BINSEQUENCE [110]. The reason for this comparison is to verify the efficiency of the first two stages of detection prior to the third stage of fuzzy matching, as BINSEQUENCE employs fuzzy matching approach after a pre-filtering process. In this experiment, we compare three different versions of `zlib` library (v1.2.5, v1.2.6, v1.2.7) with their next version using BINARM with the same setup performed in BINSEQUENCE. For instance, we test `zlib` v1.2.5 against its successive version `zlib` v1.2.6 together with two million noise functions in the database. We collect the search time for each scenario, and obtain the average time of 0.0002 seconds per function as reported in Table 5.9. On the other hand, the average of search times for these three scenarios provided by BINSEQUENCE [110] is 0.909 seconds per function. These results confirm that BINARM is three orders of magnitude faster than BINSEQUENCE.

Qualitative Comparison with GEMINI

One of the latest iterations in code similarity detection in binaries, called GEMINI [213], extracts attributed control flow graphs and then employs `structure2vec` [59] and the `Siamese` architecture [31] in order to generate the graph embeddings of two similar functions close to each other. Since during the time of this research the tool was not publicly available, only a qualitative comparison was performed as follows. (i) The required training time of GEMINI, which is performed on a powerful server with two CPUs and one GPU card, is significant compared to BINARM. (ii) The time required to constantly retrain the neural network and re-generate the embeddings is a major disadvantage in a real-world scenario. As such, BINARM greatly outperforms GEMINI with respect to the indexing of new vulnerable functions. (iii) GEMINI has a total of 154 vulnerable functions and presents a use case that employs two of them. In contrast, BINARM's *Vulnerability Database* contains 235 vulnerable functions, all of which are used for vulnerability identification. (iv) GEMINI solely relies on a few basic features and the use of a siamese neural network to perform the comparison. Such feature choices are reflected through the reported vulnerability identification accuracy of about 82% [213]. In contrast, BINARM's much richer collection of features and the rigorous feature selection process help to obtain a 92% accuracy. This is partially due to the fact that BINARM takes into account a much broader scope of information relative to a given function.

5.5.5 Detecting Vulnerabilities in Real Firmware

In this section, we demonstrate BINARM's capability to facilitate the vulnerability identification process in real-world IED firmware. We randomly select five firmware images from our *Firmware*

zlib Version	BINARM	BINSEQUENCE [110]
1.2.5	0.00057	0.897
1.2.6	0.00016	0.913
1.2.7	0.00009	0.918
Average	**0.00027**	**0.909**

Table 5.9: Baseline comparison on search time (seconds) per function

Database and compare them to all vulnerable functions in our *Vulnerability Database*. Each resulting function pair is ranked using the similarity score. We consider a candidate as a potential match, if the matching score is higher than 80%. We successfully identify 93 potential CVEs, the majority of which are confirmed by our manual analysis.

Firmware	CVE	Score	Firmware	CVE	Score
NI PMU1_0_11	CVE-2016-6303	1.00	*Schneider Link150*	CVE-2015-0208	0.68
	CVE-2014-8176	1.00	*Schneider M251*	CVE-2014-2669	0.65
	CVE-2014-6040	0.92	*ReadyNAS v6.1.6*	CVE-2015-7497	0.98
	CVE-2016-7167	0.91		CVE-2014-2669	0.97
	CVE-2015-0288	0.91		CVE-2015-7941	0.95
Honeywell.RTUR150	CVE-2016-0701	1.00		CVE-2014-6040	0.93
	CVE-2016-2105	0.99		CVE-2010-1633	0.93
	CVE-2010-1633	0.94		CVE-2014-0160	0.92
	CVE-2016-6303	0.94		CVE-2015-0288	0.91
	CVE-2015-0287	0.92		CVE-2014-3566	0.76

Table 5.10: Identifying CVEs in real-world firmware images

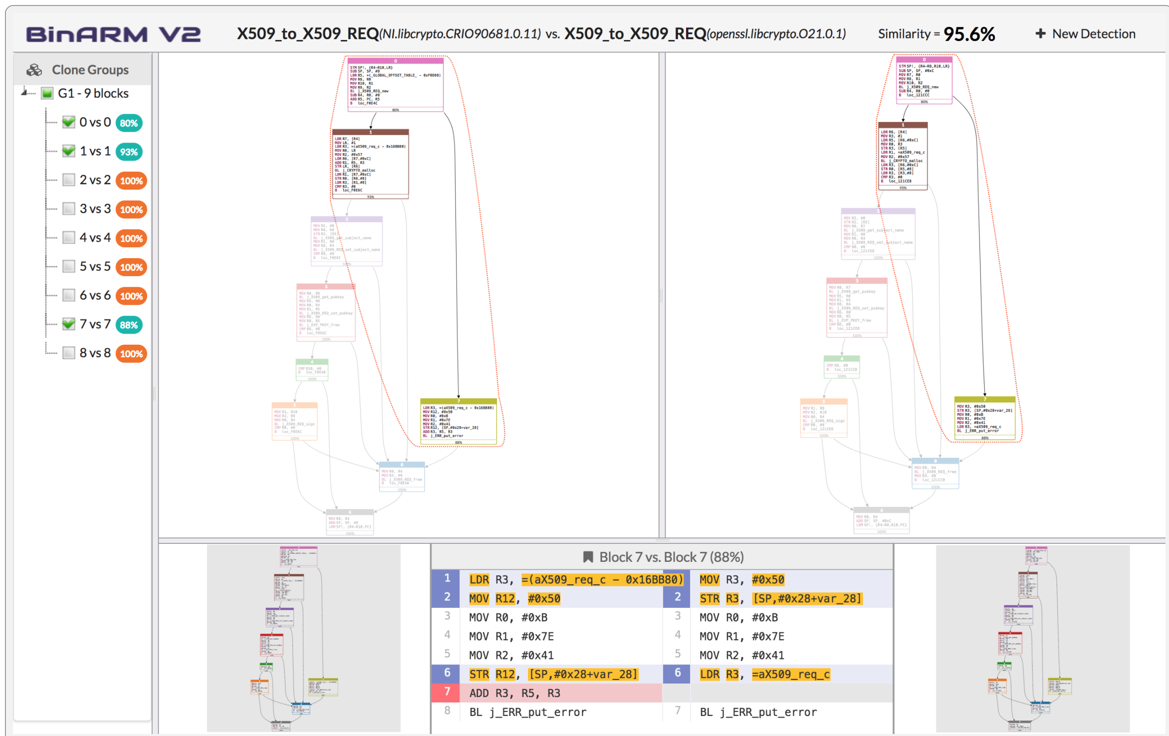


Figure 5.13: A snapshot of BINARM's in-depth vulnerability results for vulnerability search in *NI PMU1_0_11* firmware

A subset of obtained results are presented in Table 5.10. As shown, BINARM can successfully identify different vulnerabilities in the *NI PMU1_0_11*, *Honeywell.RTUR150*, and *ReadyNAS v6.1.6* firmware images. For instance, a critical heap-based buffer overflow vulnerability (CVE-2016-7167) with 0.91 similarity score is identified in *NI PMU1_0_11* firmware. The obtained matching results of vulnerable function `X509_to_X509_REQ` (CVE-2015-0288) in *NI PMU1_0_11* firmware are depicted in Figure 5.13, which illustrates BINARM's capability for providing in-depth mapping results for verification purposes. Additionally, our experiments demonstrate that BINARM can identify CVE-2014-0160 (Heartbleed vulnerability) and CVE-2014-3566 (POODLE vulnerability) in ReadyNAS firmware (as also demonstrated by the state-of-the-art approaches [78, 84]) in less than 0.5 ms. The results confirm the capability of BINARM to be applied in real-world scenarios to perform vulnerability analysis on the IED firmware embedded in the smart grid.

5.5.6 Impact of Multiple Detection Stages

In order to study the impact of the proposed multi-stage detection engine, we employ four experiments by enabling and disabling shape-based and path-based detectors (we always keep the fuzzy matching-based detector enabled), and measure both the accuracy and efficiency of BINARM on *Vulnerability Database*. To this end, we perform the tests on randomly selected projects with different versions and optimization settings. As shown in Table 5.11, the accuracy results remain the same and it has not been affected by any of the prior detection stages. On the other hand, the proposed multi-stage detection approach improves the efficiency of BINARM, as the time decreases when more detection stages are enabled.

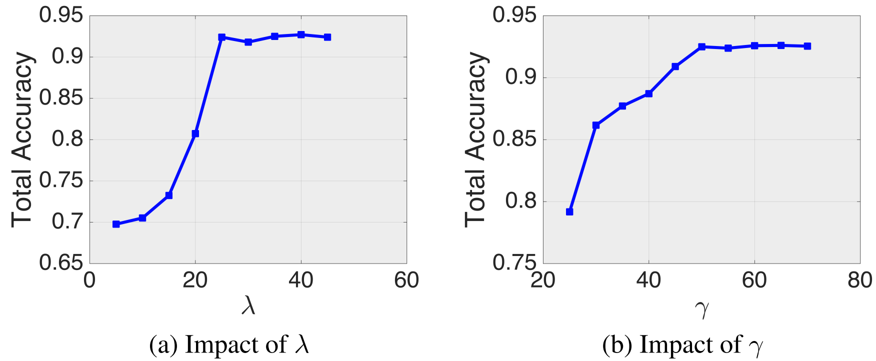
Shape-based	**Path-based**	**Accuracy**	**Time (s)**
True	True	0.929	626.72
True	False	0.928	3649.80
False	True	0.925	44823.34
False	False	0.924	50671.66

Note: The fuzzy-based detector is always enabled.

Table 5.11: Impact of detection stages

5.5.7 Impact of Parameters

In this subsection, we further study the impact of λ and γ parameters on BINARM accuracy. We perform experiments by (i) disabling the path-based detector, and incrementing the value of λ by 5 starting from an initial value of 5; (ii) disabling the shape-based detector and incrementing the value of γ by 5 each time, starting from an initial value of 25%. We randomly select 10% of libraries from our *Vulnerability Database* as the test set, and perform the matching against remaining libraries in our dataset. We repeat this process multiple times and record the accuracy. The experimental results illustrated in Figures 5.14a and 5.14b demonstrate that the values of $\lambda = 26.45$ and $\gamma = 50\%$ return the highest accuracy among other values.



(a) Impact of λ

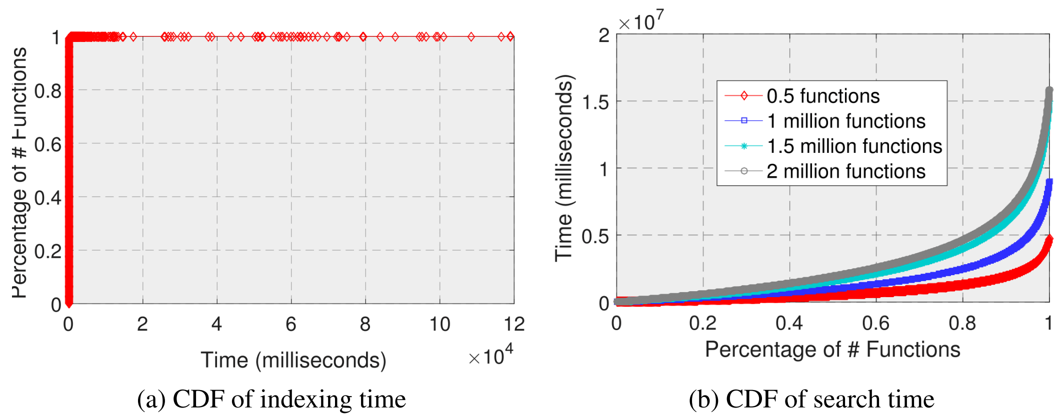
(b) Impact of γ

Figure 5.14: Impact of parameters

5.5.8 Scalability Study

We further investigate the time required for both indexing and retrieving matched functions to demonstrate BINARM capability to handle firmware analysis at a large scale. To this end, we randomly index one million functions from the *General Dataset*, and collect the indexing time per function. Figure 5.15a depicts the CDF of the preparation time for the randomly selected functions, and most of the functions are indexed in less than 0.1 second, where the median indexing time is 0.008 seconds, and it takes 0.02 seconds on average to index a function.

Moreover, we perform several scalability benchmarks, each utilizing a randomly selected set of $10,000$ target functions. For each evaluation, we employ a randomly selected set of reference



(a) CDF of indexing time

(b) CDF of search time

Figure 5.15: Scalability study

functions, where its size increases in increments of 0.5 up to 2 million, as shown in Figure 5.15b.

5.6 Limitations and Concluding Remarks

In this chapter, we presented BinArm, a scalable and efficient vulnerability detection technique for smart grid IED firmware. We proposed two substantial databases of smart grid firmware and relevant vulnerabilities. We then introduced a multi-stage detection engine that leveraged this data and identified vulnerable functions in IED firmware accurately and efficiently. This was further ramified by its evaluation on real-world IED firmware images which resulted in the identification of 93 potentially vulnerable functions.

However, BinArm has the following limitations. (i) We do not currently support function inlining. This problem can be circumvented by leveraging data flow analysis to our multidimensional fingerprint. We will study how to systematically address this problem in future work. (ii) Our proposed system deals with ARM hardware architecture. We chose the ARM architecture because most of IEDs embedded in industrial control systems are based on ARM processors. We will study how to adopt BinArm to different architectures in our future work. (iii) We currently do not consider type inference in our proposed features. However, type information is important to mitigate some sort of vulnerabilities [35]. For instance, buffer overflow exploitation can be prevented by rewriting variable type information executables [197].

Chapter 6

Code Similarity Detection in Cross-Architecture Obfuscated Binaries

Today's Internet of Things (IoT) environments are heterogeneous as they are typically comprised of devices equipped with various CPU architectures and software platforms. Therefore, in defending IoT environments against security threats, the capability of cross-architecture vulnerability detection in firmware images is of paramount importance. In this chapter, we propose TIOHTIÀ:KE, a deep learning-based approach for code similarity detection in IoT firmware images or binaries that are possibly obfuscated and obtained through different compilers for various architectures. A key idea that guides our research is the analogy drawn between the translation in several natural languages, and code similarity of functions written in different assembly languages representing different architectures.

This chapter is organized as follows. The cross-architecture code similarity detection problem is discussed in Section 6.1. The approach overview is presented in Section 6.2. The details of our vulnerability database and function representation are described in Section 6.3 and Section 6.4, respectively. Code similarity detection approach and the evaluation results are presented in Section 6.5 and Section 6.6, respectively. The preliminary study towards our future research and concluding remarks are presented in Section 6.7.

6.1 Introduction

We have been witnessing a massive adoption and deployment of IoT devices in different sectors, such as healthcare, transportation, industrial control systems, retail, smart cities, and home networks. For instance, the number of connected IoT devices is estimated to reach more than 41.6 billion by 2025[1]. Another study[2] shows that by 2025 the global IoT market is expected to grow to a value of USD 1256.1 billion. These devices enable a plethora of new services and applications with their sensing and control capabilities. Nevertheless, such deployment induces severe and challenging security concerns, especially when it comes to critical infrastructure. In addition, the increasing complexity and diversity introduced by these technologies amplify the possibility of design and implementation flaws in these systems.

More specifically, IoT devices are popular subjects to different cybersecurity threats mainly due to their Internet connectivity, complex design and diverse environments. For instance, state-backed IoT malware show that targeted attacks on IoT devices can evade traditional cybersecurity detection and cause catastrophic failures with significant impact to critical infrastructure. Examples include *Industroyer* (also referred to as *CRASHOVERRIDE*) [48, 196] targetting Ukraine's power grid to control substation switches and circuit breakers, and *BlackEnergy* [164] against the Ukranian's train railway and electricity generation utilities. Moreover, according to the study conducted by Lloyd's and the University of Cambridge's Centre for Risk Studies [79], a large-scale cyberattack can lead to a $243 billion to $1 trillion loss to the U.S. economy. Such attacks are sometimes caused by implementation flaws and vulnerabilities in the embedded software or device firmware images. However, identifying vulnerabilities in IoT firmware images and binaries is a challenging task mainly due to the use of: (i) various CPU architectures, (ii) obfuscation techniques, and (iii) different compilers and optimization settings.

Lately, we have witnessed a surge of interest in designing and implementing techniques for function similarity on binary executables and firmware images. Some of these techniques use machine learning and deep neural networks [151, 213], natural language processing [69], and graph

[1] https://www.idc.com/getdoc.jsp?containerId=prUS45213219. Accessed on Dec 20, 2020.
[2] http://bit.ly/3hkVg5z. Accessed on Dec 20, 2020.

theory [110]. Indeed, function similarity is a capability that enables a plethora of use cases, such as reverse engineering and threat analysis, library function identification [63], vulnerability detection and bug search [174], and authorship attribution [7].

The genesis of our research is stemming from two important considerations:

- Most of the existing contributions in terms of function similarity are targeting a single architecture (e.g., [69, 11, 189]). As such, there is a desideratum that consists of elaborating an accurate, scalable, and efficient technique for cross-architecture function similarity. Such a capability is highly needed considering the current landscape of platforms and also the high diversity of the deployed IoT devices. If elaborated, such technique will enable the aforementioned applications.
- A few works (e.g., [69, 118]) perform function code similarity in the presence of code transformation techniques (e.g., obfuscation). However, these works support solely a single architecture. When it comes to the applications of code similarity, code transformation is an unavoidable reality nowadays. Indeed, most of binaries and firmware images are generated with different CPU architectures, and very often employ obfuscation in the case of malware threats or intellectual property protection, in order to deter or impede reverse engineering.

Therefore, the primary objective of this research is to design and implement a cross-architecture code similarity detection in the presence of code transformation techniques, e.g., code optimization and obfuscation. A significant sub-goal of our research is to build upon function similarity to design an automatic cross-architecture and cross-compiler vulnerability detection in obfuscated binaries and firmware images. Such a capability is of paramount importance as it will support the security assessment of IoT devices prior to their deployment.

In essence, most of the existing proposals suffer from one of the following limitations: (i) support of a single architecture, (ii) lack of resilience to obfuscation techniques and code transformation, and (iii) not focusing on large-scale vulnerable function detection. In this research, we address the aforementioned limitations by proposing TIOHTIÀ:KE, a code similarity detection approach that supports cross-architecture obfuscated binaries. To demonstrate TIOHTIÀ:KE's applicability to the

security assessment of real-world IoT devices, we first build a large-scale vulnerability database and then instantiate TIOHTIÀ:KE for the detection of vulnerable functions in real-world firmware images acquired from smart grid intelligent electronic device (IEDs).

More specifically, we employ neural machine translation (NMT) by leveraging the similarity between assembly languages and the translation between natural languages. In our context, we consider each function representation as a sentence and translate similar functions from a source language (e.g., ARM) into a target language (e.g., x86). To this end, we first lift the disassembeld functions into an intermediate representation to remove the effects of different CPU architectures. Second, we model function representations with obfuscation-resilient features and a sequence of execution paths. Finally, we learn function embeddings and search for similar functions by leveraging an LSTM Encoder-Decoder architecture [105], which is a popular choice for various purposes, such as language translation, text summarizing and chat bots.

Contributions. Our main contributions are as follows:

- To the best of our knowledge, this is the first approach that performs code similarity detection in cross-architecture binaries in the presence of different obfuscations and optimizations. For this purpose, we leverage the power of neural machine translation (e.g., LSTM Encoder-Decoder) models to translate semantically equivalent functions from a source CPU architecture to a target CPU architecture.

- We introduce a new function representation by identifying new features that are less affected by code transformation techniques (e.g., compiler and obfuscation), such as *call walks* in tandem with *edge coverage*. Further, the new function representation involves function summarizing in order to overcome the memory constraints of LSTM models.

- We build the first large-scale cross-architecture cross-compiler obfuscated vulnerability database in the context of smart grid. It includes about 9 million functions mainly from the top-25 identified open-source libraries reused in the intelligent electronic devices of the smart grid, among which $14,967$ are vulnerable.

- We perform extensive experiments using both vulnerability database and obfuscated datasets

to demonstrate the accuracy and efficiency of our proposed approach. Obtained results confirm the out-performance of TIOHTIÀ:KE over state-of-the-art approaches [69, 151]; accuracy improvement up to 0.993 for code similarity detection and obtaining the recall of 0.903 for obfuscated code. Also, we show that our solution can effectively identify CVEs in real-world IoT device firmware images.

6.2 Approach Overview

In order to perform code similarity detection, we propose a technique based on neural machine translation and sequence-to-sequence learning. The proposed solution learns the translation of a given function from one architecture (e.g., x86) to another architecture (e.g., ARM). The overview of our approach is depicted in Figure 6.1. Our approach consists of three phases: *dataset generation*, *function representation*, and *learning and inference*, as explained in the following.

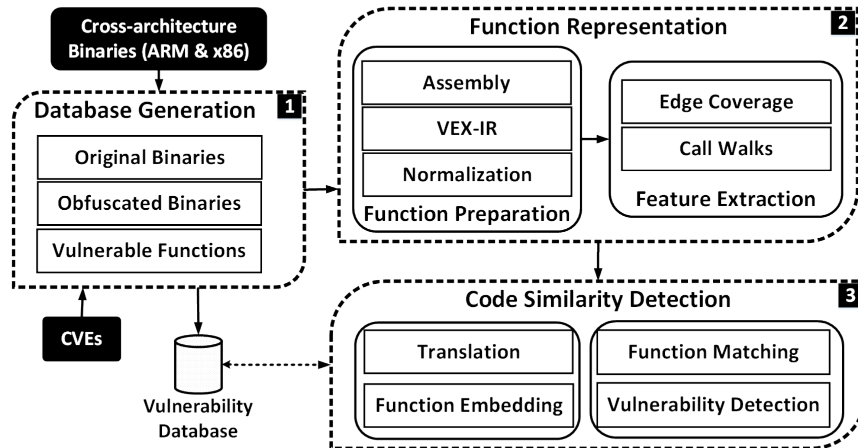


Figure 6.1: TIOHTIÀ:KE approach overview

- *Database generation.* We build a large repository of open-source libraries that reflect different obfuscations, compiler families, optimization settings and CPU architectures. The selection of open-source libraries is based on those that are found in IEDs in the smart grid context. We further label the vulnerable functions by cross-referencing them with the CVE database[3].

[3] https://cve.mitre.org/. Accessed on Dec 20, 2020.

- *Function representation.* We lift the disassebmled functions into VEX intermediate representations (IR) [166] and further normalize them to reduce the effects of different CPU architectures and compilers. We then extract the edge coverage and call walks from the normalized VEX representation of the functions to capture function execution paths and cover obfuscation-resilient features. Finally, we model function representations as a sequence, which is composed of the combination of edge coverage and call walks.

- *Code similarity detection.* We leverage the LSTM Encoder-Decoder [49], a neural machine translation technique, to perform code similarity detection. More specifically, we train our neural model to translate a given function from one architecture (e.g., x86) into its equivalent version for another architecture (e.g., ARM). We detect vulnerabilities in a given function by matching with vulnerable functions in our vulnerability database.

We elaborate on each of the phases in the following.

6.3 Vulnerability Database Generation

We build a large dataset containing different popular open-source libraries, which are frequently used in the firmware images of smart grid IEDs. To this end, as in Chapter 5, we first identify the manufactures that produce devices for the smart grid applications. We identified 14 manufactures, including Schweitzer Engineering Laboratories (SEL) and General Electrics (GE). Then, we utilize popular FTP search engines and create a simple website scraper to retrieve firmware images that are relevant to the identified vendors. We build a firmware database consisting of $2,687$ firmware images. Afterwards, we investigate several sources of information pertaining to identified manufacturers to extract large amounts of open-source declarations. These sources include corporate websites, product documentations, download centers, and FTP search tools. Next, we identify publicly known CVEs for each of these libraries by utilizing the CVE database. Afterwards, we order them by their weights, which are obtained by multiplying the number of CVEs, for a given library, to the number of manufacturers that utilize that library. We consider the top-25 relevant, vulnerable and reused open-source libraries. The obtained results are illustrated in Table 6.1, in terms of the

numbers of CVEs reported in both 2017 (from our work in Chapter 5) and in 2020. As can be seen, the number of CVEs has increased in the majority of identified libraries.

Library Name	**Compiled**	**#CVEs** (2017)	**#CVEs** (2020)	**Manufacturers**					
				ABB	Cisco	GE	Honeywell	SE	Siemens
`php`		601	↗ 626		•		•		•
`imagemagick`	✓	402	↗ 546		•	•	•		
`openssl`	✓	189	↗ 204	•	•	•	•	•	•
`mysql`	✓	223	223		•				
`tcpdump`		162	↗ 171		•	•			•
`openssh`	✓	87	↗ 98	•	•	•	•		•
`ntp`	✓	79	↗ 88		•	•	•	•	•
`libtiff`		149	↗ 174		•	•			
`postgresql`	✓	98	↗ 112		•		•		•
`ffmpeg`		247	↗ 316						•
`pcre`		47	↗ 48	•	•	•	•		•
`python`		32	↗ 49		•		•		•
`glibc`	✓	81	↗ 108		•		•		•
`qemu`		225	↗ 276		•				
`libxml2`	✓	45	↗ 65	•	•	•	•		•
`bind`		102	↗ 119		•				•
`binutils`	✓	97	↗ 179		•				•
`libcurl`	✓	5	5	•	•		•	•	•
`freetype`		76	↗ 80		•				•
`libpng`	✓	33	↗ 40		•		•		•
`samba`		124	↗ 157				•		
`utillinux`		4	4	•	•	•	•	•	•
`cups`		8	↗ 10		•				
`lighttpd`	✓	28	↗ 29	•	•		•		
`netsnmp`	✓	20	↗ 23		•	•		•	•

Table 6.1: Top-25 identified vulnerable open-source libraries used in smart grid IEDs

Furthermore, we extend our vulnerability database by adding more variations of the binaries, which are originated from several architectures, compilers and obfuscation techniques. More specifically, we consider the OBFUSCATOR-LLVM [120] for obfuscation purposes, which performs *instruction SUBstitution* (SUB), *Bogus Control Flow* (BCF), and *control flow FLAttening* (FLA) obfuscation techniques. *Instruction substitution* replaces the instructions with functionally equivalent but more complicated sequences of instructions. The *bogus control flow* techniques insert new basic blocks that contain an opaque predicate and then make a conditional jump back to the original basic block. *Control flow flattening* flattens the control flow graph, which first splits the body of function into basic blocks and then puts all the blocks at same level (e.g., using a `switch` statement).

PROPOSAL	Dataset				Compiler				Arch.			OS	
	Original Binaries	Obfuscated Binaries	Firmware Images	Vulnerable Functions	VS	GCC	ICC	Clang	x86-64	ARM	MIPS	Window	Linux
MULTI-MH [174]	60	0	4	4		•		•	•	•	•	•	•
BINGO [41]	110	0	NA	2		•		•	•	•		•	•
ESH [60]	1,000	0	NA	8		•	•	•	•				•
DISCOVRE [78]	2,280	0	2	3	•	•	•	•	•	•	•	•	•
GENIUS [84]	17,626	0	8,128	154		•		•	•	•	•		•
GITZ [61]	~500K	0	NA	9		•	•	•	•	•			•
GEMINI [213]	51,314	0	8,126	154		•			•	•	•	•	
ASM2VEC [69]	16	1,116	NA	8		•	•	•	•				•
SAFE [151]	11	0	NA	8		•		•	•	•			•
BINARM [189]	1,174	0	5,756	5,103		•				•			•
TIOHTIÀ:KE	1,208	6,300	5,831	14,967		•		•	•	•			•

(•) means that the approach provides the corresponding feature, it is empty otherwise. (NA) means that the corresponding work dose not consider firmware analysis.

Table 6.2: A comparison between the vulnerability databases of TIOHTIÀ:KE and state-of-the-art approaches

We collect different source code samples of top-25 identified libraries as well as additional libraries (e.g., `libgmp`) for comparison purposes from their official websites. We then cross-compile them for the ARM and x86 architectures using *GCC* compiler with different optimization settings, i.e., `O0-O3`. Moreover, we leverage OBFUSCATOR-LLVM to generate the corresponding obfuscated versions for a subset of libraries in our database using the *clang* compiler with different optimization settings and obfuscation techniques, i.e., FLA, BCF and SUB. Then, we identify and label known CVEs for each library by utilizing the CVE database. Our vulnerability database consists of about 9 million functions, 14,967 of which are labelled as vulnerable including the duplicates resulting from various CPU architectures, compilers, optimization settings and obfuscations.

Comparison. We compare our vulnerability database with the vulnerability datasets that are used in the state-of-the-art approaches. The summary of this comparison is shown in Table 6.2. As seen, TIOHTIÀ:KE has the largest vulnerability database for binaries compiled for x86 and ARM architectures. The GENIUS [84] and GEMINI [213] approaches have less vulnerable functions, although they include the additional architecture of MIPS. Moreover, only one other work, i.e., ASM2VEC [69] generates 1,116 obfuscated binaries (including different obfuscation techniques) whereas TIOHTIÀ:KE builds a repository of 6,300 obfuscated binaries.

6.4 Function Representation Generation

This section proposes a function representation method, output of which will later be the input to the LSTM encoder-decoder model. We elaborate on each step for function representation as follows.

6.4.1 Intermediate Representation

In order to reduce architecture-specific dependencies, we employ an existing lifter to translate binary codes to an intermediate representation (IR). We leverage VEX-IR, which is the well-known lifter of VALGRIND project [166] and is re-implemented in the PYVEX [192] repository of ANGR [193] framework. Our choice of VEX-IR is mainly motivated by its wide well-documented active repository and its support for different CPU architectures (e.g., x86, ARM, and MIPS). The VEX-IR abstracts away from several architecture differences, such as register names, instruction side effects, memory accesses, and memory segmentations. The VEX intermediate representation mainly consists of *expression* and *statement* components. The *expressions* represent a calculated or constant value (e.g., register reads), while the *statements* reflect the changes in the state of a target machine (such as updating a memory location) that may take the *expressions* as input.

Data Pre-processing and Normalization. Since two *expressions/statements* might be identical, but differ in terms of memory references or temporary variables, it is essential to normalize them for both code similarity detection and neural machine translation models (Section 6.5.2). To this end, we propose to normalize *expressions* and *statements* such that their syntax differences do not affect the code similarity detection results. An excerpt of the proposed normalization for VEX *expressions* and *statements* is presented in Table 6.3 and Table 6.4, respectively. For instance, the temporary variables (e.g., `t10`) are normalized to `TMP`; and the `GET:I32(0)` *expression* that gets the value of 32-bit `%eax` register (`%eax` is indicated by the value of 0) will be replaced with `GET:IX(REG)`.

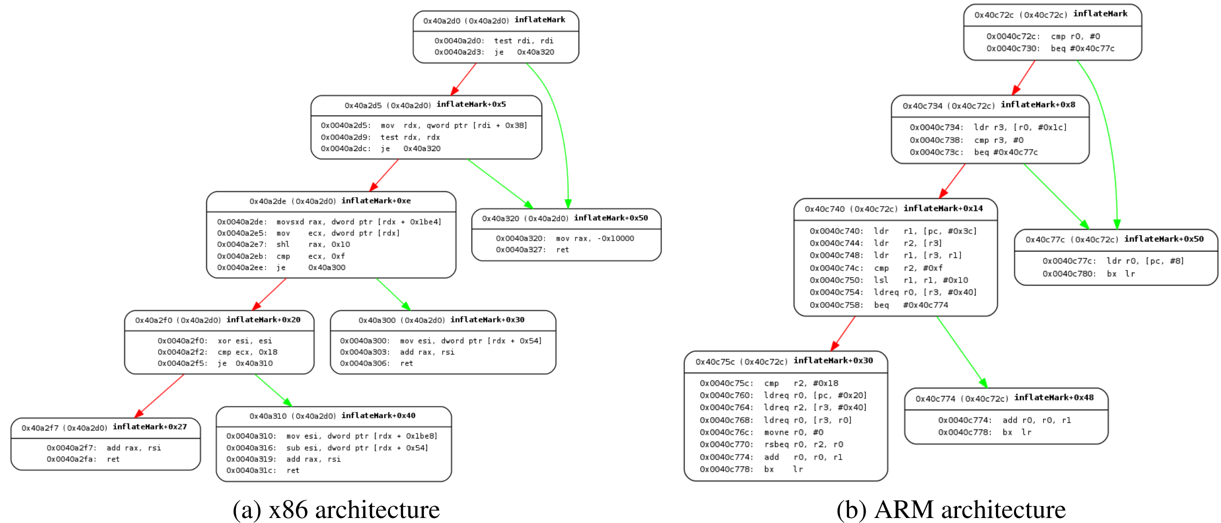
Example 6.4.1 *In the following, we illustrate an example of the disassembled CFGs in ARM and x86 architectures and their equivalent versions in VEX-IR. We compile* `zlib` *v1.2.8 library with GCC compiler and optimization setting* `O2` *for both x86 and ARM architectures. The assembly*

IR Expression	VEX Output Example	Normalized Output
Read Temp	`RdTmp(t10)`	`RdTmp(TMP)`
Get Register	`GET:I32(16)`	`GET:IX(REG)`
Load Memory	`LDle:I32\LDbe:I64`	`LDle:IX`
Operation	`Add32`	`AddX`

Table 6.3: Proposed normalization for IR *expressions*

IR Statement	VEX Output Example	Normalized Output
Write Temp	`WrTmp(t1) = (Expression)`	`WrTmp(TMP)=(Expression)`
Put Register	`PUT(16) = (Expression)`	`PUT(REG)=(Expression)`
Store Memory	`STle(0x1000) = (Expression)`	`STle(MEM)=(Expression)`
Exit	`if (condition)`	`if(condition)`
	`goto (Boring) 0x4000A00:I32`	`goto(Boring) MEM:IX`

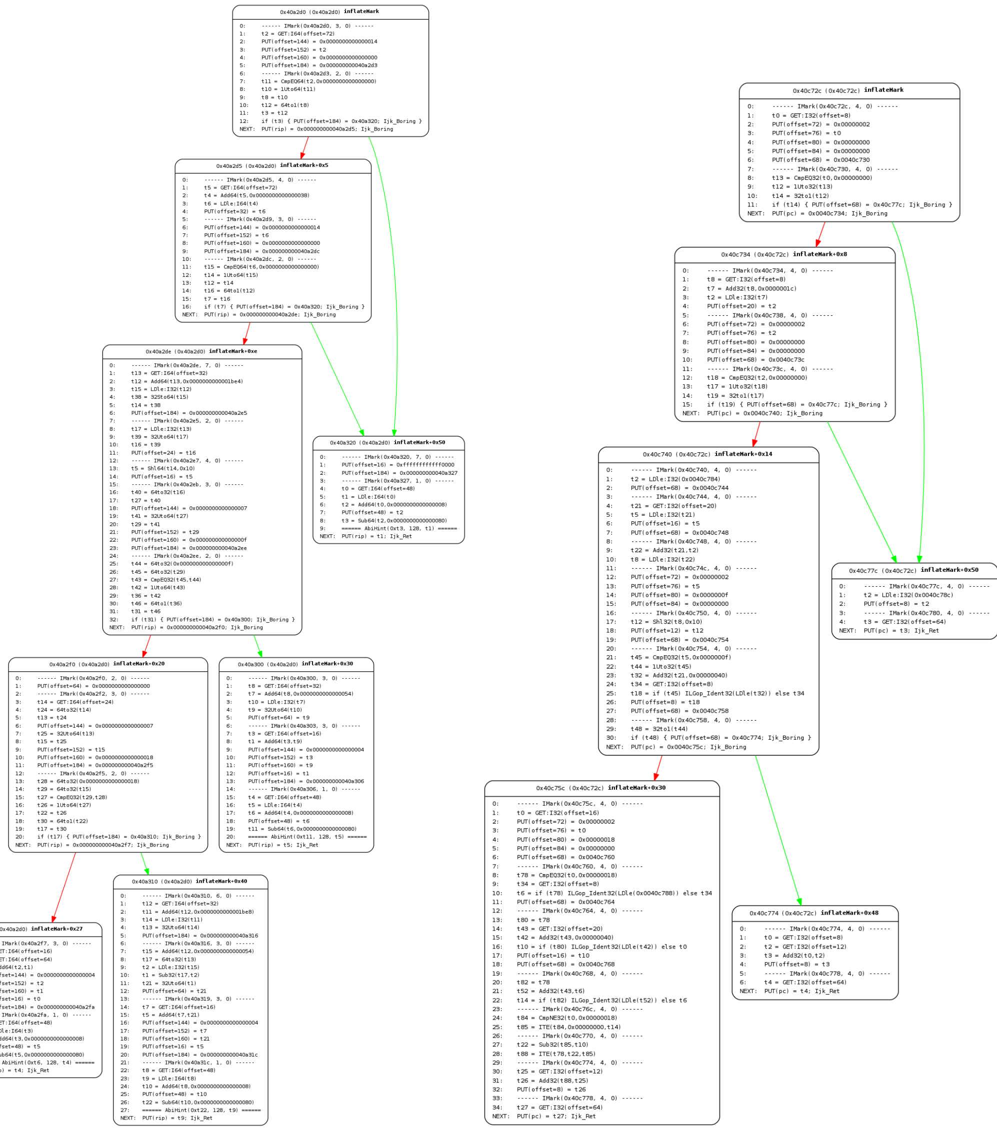
Table 6.4: Proposed normalization for IR *statements*



(a) x86 architecture (b) ARM architecture

Figure 6.2: CFGs of `zlib`-v1.2.8-`inflateMark` function compiled with GCC-`O2` compiler for x86 and ARM architectures

representations of `inflateMark` *function in x86 and ARM architectures are illustrated in Figures 6.2a and 6.2b, respectively. As can be seen, the assembly instructions are completely different, while the control flow graphs are relatively similar. We employ VEX-IR and get the corresponding IR representations as depicted in Figures 6.3a and 6.3b. The first basic block of* `inflateMark` *function and its equivalent VEX-IR as well as normalized VEX-IR representations in both x86 and*



(a) x86 architecture (b) ARM architecture

Figure 6.3: VEX-IR of `zlib`-v1.2.8-`inflateMark` function illustrated in Figure 6.2

ARM architectures are represented in Figure 6.4. As seen, the obtained normalized VEX representations are very similar to each other. The only difference is in two normalized values while the original assembly instructions are syntactically totally different.

```
------------ x86 - Assembly -----------
test rdi, rdi
je 0x40a070
```

```
------------ ARM - Assembly -----------
cmp r0, #0
beq #0x409180
```

```
------------ x86 - VEX-IR ------------
t2 = GET:I64(offset=72)
PUT(offset=144) = 0x0000000000000014
PUT(offset=152) = t2
PUT(offset=160) = 0x0000000000000000
PUT(offset=184) = 0x000000000040a2d3
t11 = CmpEQ64(t2,0x0000000000000000)
t10 = 1Uto64(t11)
t8 = t10
t12 = 64to1(t8)
t3 = t12
if (t3) {
        PUT(offset=184) = 0x40a320;
        Ijk_Boring
}
```

```
------------ ARM - VEX-IR -----------
t0 = GET:I32(offset=8)
PUT(offset=72) = 0x00000002
PUT(offset=76) = t0
PUT(offset=80) = 0x00000000
PUT(offset=84) = 0x00000000
PUT(offset=68) = 0x0040c730
t13 = CmpEQ32(t0,0x00000000)
t12 = 1Uto32(t13)
t14 = 32to1(t12)
if (t14) {
          PUT(offset=68) = 0x40c77c;
          Ijk_Boring
}
```

```
------ x86 - Normalized VEX-IR ------
TMP=GET:IX(REG);
PUT(REG)=CONST;
PUT(REG)=TMP;
PUT(REG)=CONST;
PUT(REG)=CONST;
TMP=CmpEQX(TMP,CONST);
TMP=XUtoX(TMP);
TMP=TMP;
TMP=XtoX(TMP);
TMP=TMP;
if(TMP);
{;
        PUT(REG)=MEM;
        Ijk_Boring;
};
```

```
------ ARM - Normalized VEX-IR ------
TMP=GET:IX(REG);
PUT(REG)=CONST;
PUT(REG)=TMP;
PUT(REG)=CONST;
PUT(REG)=CONST;
PUT(REG)=CONST;
TMP=CmpEQX(TMP,CONST);
TMP=XUtoX(TMP);
TMP=XtoX(TMP);
if(TMP);
{;
        PUT(REG)=MEM;
        Ijk_Boring;
};
```

Figure 6.4: First VEX-IR basic block in `zlib`-v1.2.8-`inflateMark` function

6.4.2 Feature Engineering

In this section, we describe our choices of features that contain function semantics and are more resilient to code transformation techniques (that are utilized in this work). TIOHTIÀ:KE captures function semantics through natural language processing techniques, which are capable of learning semantic relationships amongst different words in a given context. As such, we aim at modeling functions with a sequence of words/tokens to learn their contextual semantic relationships and to translate functions from one CPU architecture to another one using LSTM Encoder-Decoder model. Representing functions with a sequence of tokens can be performed by considering the whole instructions, i.e., IR expressions/statements, in a sequential order. However, different code

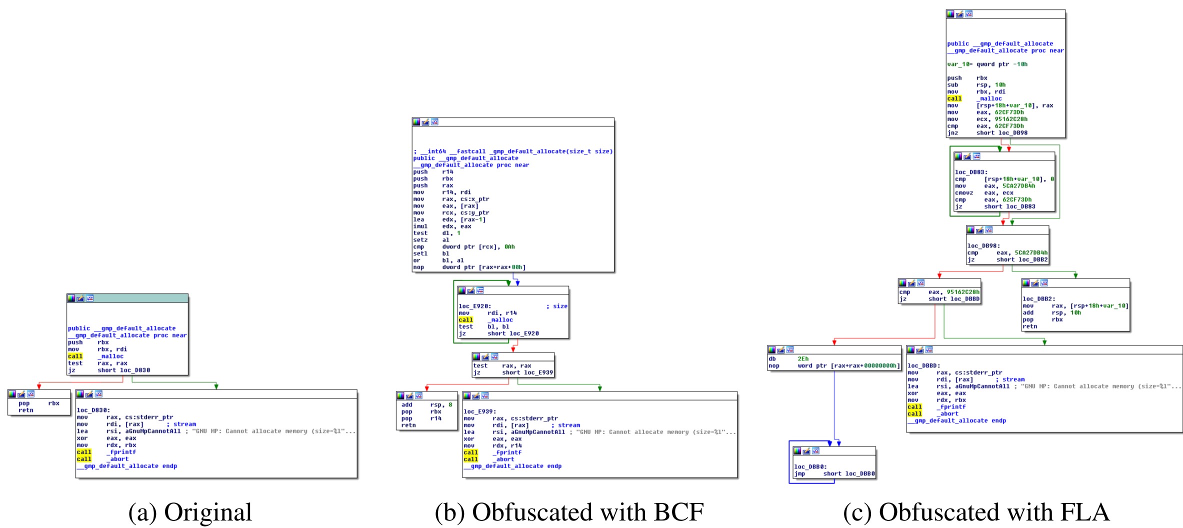
transformation techniques will affect the instructions orders. To overcome this effect, we extract *edge coverage* to model functions as a sequence.

Furthermore, code transformation techniques, such as obfuscation, might inject dummy basic blocks, which is not related to the main functionality of a function. To reduce obfuscation effects, we propose to enrich function representations with *call walks* that encompass solely the nodes containing system and function calls. We represent each function by concatenating its edge coverage and call walks. According to our experiments, these two features can represent a function for accurate code similarity detection. In the following, we explain these features in more details.

Call Walks

The obfuscation techniques, such as BCF and FLA, usually insert additional instructions and basic blocks compared to the original function in order to thwart code analysis. However, according to our statistical analysis, system and function calls will be resilient to obfuscation techniques to a large extent. In fact, system calls are an essential part of code semantics and their excessive alteration would tamper the original functionality. Accordingly, we propose to use *call walks* as part of function representation to help reduce the effects of obfuscation techniques under study in this work.

Example 6.4.2 *Consider the original `libgmp` v6.1.0 library cross-compiled for x86 and ARM architectures with clang compiler (`O2`) and its obfuscated versions using the LLVM obfuscator. The original CFG of `default-allocate` function and the corresponding BCF and FLA CFGs are illustrated in Figure 6.5 and Figure 6.6 for x86 and ARM architectures, respectively. As seen, the FLA technique introduces additional number of blocks which is usually significant in relation to the function size. These blocks mainly contain one or two instructions including compare (e.g., `cmp`) and branch (e.g., `jmp`) instructions. Similarly, BCF techniques introduce new basic blocks, which are not part of the main functionality. While the instructions and the structure of the function are changed, in all of these representations, the libc calls (e.g., `_malloc`, `_fprintf` and `_abort`) remain the same. Noteworthy, these libc calls are located in the main basic blocks, e.g., the first and last blocks of the flattened functions as shown in Figure 6.5c and Figure 6.6c.*



(a) Original (b) Obfuscated with BCF (c) Obfuscated with FLA

Figure 6.5: CFGs of `default-allocate` function in `libgmp` 6.1.0 compiled with *clang*-`O2` for the x86 architecture

(a) Original (b) Obfuscated with BCF (c) Obfuscated with FLA

Figure 6.6: CFGs of `default-allocate` function in `libgmp` 6.1.0 compiled with *clang*-`O2` for the ARM architecture

We identify basic blocks that contain calls (we call them *essential blocks*) and extract only walks that contain those *essential* blocks. Our strategy is based on walking through the blocks and capture function's interactions with the operating system. If the current block is an *essential* block, we consider this block and one block after it as part of the call walks. The main reason

behind considering one block after the *essential node* is that calls in VEX-IR are converted to jump instructions. As such, occurrence of a call in the middle of a block at assembly level results in two blocks at the VEX-IR level, where the second block contains the effects of that call.

Algorithm 5: Call walks

```
   Input: graph // Normalized VEX-IR control flow graph
   Output: callWalkList // Call Walk of a function
1  Function Main(graph):
2      removeLoops(graph);
3      tree ← constructTree(graph);
4      callWalkList ← {}; callWalk ← {};
       // Passing three pointers to getCallWalks() function
5      getCallWalks(tree.root, callWalk, callWalkList);
6      return callWalkList;
7  Function getCallWalks(node, callWalk, callWalkList):
8      callWalk.append(node);
9      if node is critical then
10         if len(node.children)==0 then
11             callWalkList.append(callWalk);
12             return;
13         end
14         foreach child ∈ node.children do
15             getCallWalks (node.child, callWalk, callWalkList);
16         end
17     else
18         if previous node is not critical then
19             callWalkList.append(callWalk);
20             return;
21         else
22             if len(node.children)==0 then
23                 callWalkList.append(callWalk);
24                 return;
25             end
26             foreach child ∈ node.children do
27                 getCallWalks (node.child, callWalk, callWalkList);
28             end
29         end
30     end
31 return
```

To acquire *call walks*, we extract all the paths from a normalized VEX CFG, and search for *essential nodes* within each path. An *essential node* is defined as a basic block that contains at least one of the following symbols: {`Ijk_Sys_syscall`, `Ijk_Sys_sysenter`, `Ijk_Sys_int32`, `Ijk_Sys_int128`,

`Ijk_Sys_int129`, `Ijk_Sys_int130`, `Ijk_Sys_int145`, `Ijk_Sys_int210`, `Ijk_Ret`, `Ijk_Call`}. While walking through each path, we extract the *essential nodes* and one node after them (if any), and finally concatenate all the call walks. The details are presented in Algorithm 5.

Edge Coverage

To account for all possible reachable paths in a control flow graph, we consider *edge coverage*. Recall that in software testing [99], *edge coverage* is used to ensure that each decision condition from every branch (e.g., edge) is exercised at least once. As such, we choose to consider *edge coverage* to model function representation as in [69]. More specifically, we iterate over all edges in a normalized VEX control flow graph and concatenate the contents of corresponding basic blocks. Consequently, dominant basic blocks will occur more frequently in this new function representation. The details of the algorithm are presented in Algorithm 6. In each iteration (Line 2), the contents of basic blocks, that are connecting the current visited edge, are concatenated as a representation of an edge (Line 5). Finally, the concatenated basic blocks are considered as the representation of a function *edge coverage*.

Algorithm 6: Edge coverage

```
  Input: graph // Normalized VEX-IR control flow graph
  Output: edgeCoverage // Edge coverage of a function
1 edgeCoverage ← {};
2 foreach edge ∈ graph.Edges do
      // Get the contents of source and target nodes
3     sourceNode ← basicBlocks[edge.Source];
4     targetNode ← basicBlocks[edge.Target];
5     seq ← (sourceNode || targetNode);
6     edgeCoverage ← edgeCoverage ∪ seq;
7 end
8 return edgeCoverage;
```

6.5 Code Similarity Detection

The key idea of our code similarity detection solution is based on the analogy between assembly languages generated from different CPU architectures and different natural languages, and the possibility of applying neural machine translation techniques to this problem. As such, we utilize the LSTM Encoder-Decoder model [49] that proves its accurate results when it comes to natural language translation, text summarizing and chat bots. The LSTM Encoder-Decoder architecture is composed of two main components: *encoder* and *decoder*. The encoder summarizes a variable-length source sequence (e.g., a function in ARM) into a fixed-length vector representation, which is called *context vector*. Then, the decoder decodes the given context vector back into a variable-length target sequence (e.g., a function in x86). In fact, the encoder and decoder are jointly trained to maximize the conditional probability of a target sequence, given a source sequence, where the source and target sentences may differ in length. As such, the contextual semantic relationships between the words in a sentence are taken into account.

The major steps of our proposed LSTM encoder-decoder based function detection framework are illustrated in Figure 6.7. First, TIOHTIÀ:KE prepares binary functions from different architectures to the input format of LSTM models. Second, it learns to translate and infer function representations between different CPU architectures through LSTM encoder-decoder. Finally, it matches the output (translated function) with a set of known functions in our vulnerability database for code similarity detection.

6.5.1 LSTM Encoder-Decoder

In the LSTM Encoder-Decoder model, the *encoder* and *decoder* are implemented as LSTM networks, as shown in Figure 6.7. This model has two phases: *learning* and *inference*. The encoder has the same role in both training and inference, while the decoder has different roles. As shown, the encoder reads the whole input sequence one word at a time and generates a summary of the input sentence, i.e., context vector **c**, which is the final hidden state of the encoder .

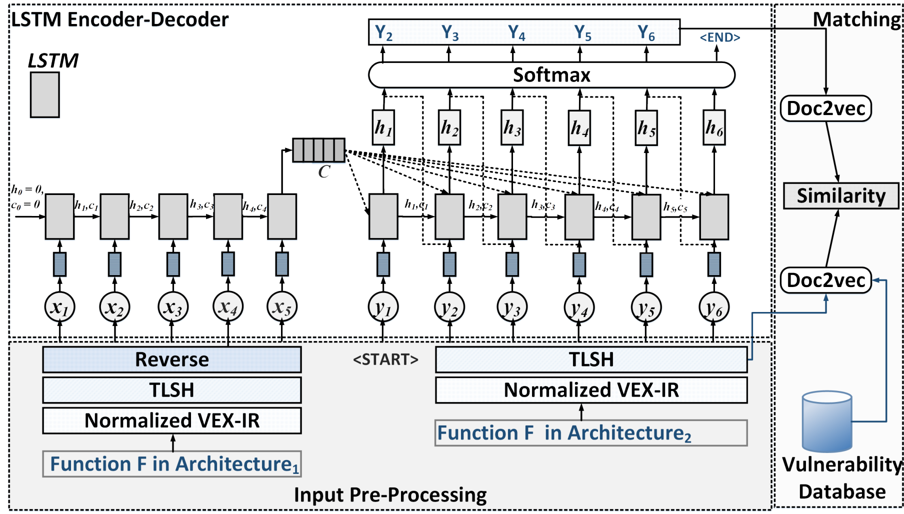


Figure 6.7: TIOHTIÀ:KE code similarity solution

During the *learning* phase, the initial states of the decoder are set to the final state of the encoder. This means that the information is provided by the encoder to start generating the output. Moreover, the input at each time step, t, is given as the actual output (and not the predicted output) from the previous time step, $t-1$ (a.k.a. *teacher forcing* [133]). This results in faster and more efficient training of the network. During the *inference* phase, the *decoder* must predict the entire output sentence given the input. At each time step, t, the predicated output is considered as the input to the next step. The *decoder* starts with the *context vector* and the `<START>` word and predicts the next word at a time. It produces the subsequent word based on the generated word and its hidden representation. It continues generating the next words until a special end-of-sentence symbol, `<END>`, is produced.

More formally, the LSTM Encoder–Decoder model is composed of the input sentence as a sequence of vectors, $\mathbf{x} = (x_1, \ldots, x_T)$, and its corresponding output sequence $(y_1, \ldots, y_{T'})$, where the length T' may differ from the length T. The goal is to estimate the conditional probability of $p(y_1, \ldots, y_{T'} | x_1, \ldots, x_T)$. The encoder reads the input sequence sequentially and the hidden states $h_t \in \mathbb{R}^n$ at time t are calculated as follows:

$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, x_t) \tag{18}$$

where f is an LSTM unit. After reading the whole input sequence, the summary of entire input sequence (final hidden state) is stored into a vector ($\mathbf{c}$).

The decoder is another LSTM that generates the output sequence by predicting the next symbol y_t given the hidden state $\mathbf{h}_t$. Both y_t and $\mathbf{h}_t$ are dependent on y_{t-1} and on the summary $\mathbf{c}$ of input sequence. Therefore, the decoder hidden state $\mathbf{h}_t$ at time t is computed as follows:

$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, y_{t-1}, \mathbf{c}) \tag{19}$$

The decoder is trained to predict the next word y_t given all the previously predicated words $\{y_1, \dots, y_{t-1}\}$ and the context vector:

$$p(\mathbf{y}) = \prod_{t=1}^{T'} p(y_t | y_1, \cdots, y_{t-1}, \mathbf{c}) \tag{20}$$

The conditional probability is defined as follows:

$$p(y_t | y_1, \cdots, y_{t-1}, \mathbf{x}) = \mathit{softmax}(g(\mathbf{h}_t)) \tag{21}$$

where g indicates the transformation function that outputs a vector.

Challenges in natural language processing. There still exists few challenges in natural language processing (NLP), such as *ambiguity* and *pseudonyms*, that might be applicable in our context too [85, 156]. For instance, *ambiguity* relates to the sentences that have multiple alternative interpretations in human conversations. Handling these cases is very challenging in NLP [85]. However, since we deal with memory addresses and registers and perform logical operations that need to be clearly defined for machines and operating systems, we assume that we do not have these kinds of issues in our context. In our future work, we will further investigate the validity of this assumption.

6.5.2 Input Pre-Processing

To train our model, the input to the LSTM encoder-decoder model should be prepared while considering binary analysis challenges. In particular, as shown in Figure 6.7, we represent each function

by concatenating its edge coverage and call walks from the normalized VEX-IR. Then, we convert the content of each basic block to a hash value using a locality sensitive hashing technique, i.e., TLSH [171], in order to overcome the memory constraints in the LSTM models for large binary functions. Finally, we reverse the input to the LSTM encoder, as it is known to have better translation accuracy [198]. We further provide the reasons behind our proposed solutions as follows.

Challenges

In addition to the general challenges of code similarity detection for cross-architecture obfuscated binaries, we encounter several additional challenges during learning process. We summarize these challenges and the elaborated solutions to address them as follows.

Challenge 1: Sequence Length: The sequential nature of LSTM models prevents parallelization during the training time, and due to computational requirements and memory constraints handling longer sequences becomes critical [205]. Therefore, existing works typically limit the input lengths after a predefined value. For instance, in a natural language processing work [147], the maximum length of a sentence is considered to be 50 words. Similarly, the binary code similarity approach SAFE [151] limits its input length to 150 instructions. However, shortening the input size and consequently ignoring a big portion of functions will affect the accuracy of our framework; especially due to the fact that obfuscation techniques may inject a significant number of instructions into each function, which results in shifting the original instructions to the later parts of a function.

Example 6.5.1 *Consider the `ImageMagick` v7.0.6.10 `libMagickCore.7.Q16HDRI` library cross-compiled for x86 architecture and clang compiler, both with FLA obfuscation and without obfuscation. An example of the number of basic blocks and number of instructions of the randomly selected functions (sorted by the number of basic blocks) is presented in Figure 6.8. As can be seen, the number of instructions are drastically higher compared to the usual limited input length that is considered in the state-of-the-art approaches. For instance, even if a model can handle a sequence with maximum 512 words/instructions, in some cases only 25% of the listed function instructions will be considered during the learning phase. This will greatly affect the accuracy of these models.*

Solution to Challenge 1: To address this challenge, we propose to summarize functions using hashing such that each basic block will be considered as an input to the LSTM Encoder-Decoder model. Therefore, instead of considering each VEX expression/statement as a word, we regard each basic blocks as a word/token[4]. As such, we apply the Trend Micro Locality Sensitive Hash (TLSH) [171] on the content of each basic block of normalized VEX-IR CFGs. The TLSH generates the same hash value for identical or very similar basic blocks. Thus, by utilizing the TLSH values we reduce the length of our sentences in which each basic block is considered as a word. This results into covering a larger portion of functions and consequently considering more expressions/statements as a part of the sequences.

We further justify our above-mentioned solution as follows. The FLA and BCF obfuscation techniques significantly affect function representations at basic block levels. For instance, both FLA and BCF techniques inject multiple basic blocks in which there are only a few similar instructions

[4]In this work, we refer to TLSH basic blocks as *words* or *tokens* interchangeably.

function_name	#nodes	instrnum
XFontBrowserWidget	502	3737
ReadBMPImage	462	3893
ReadCMYKImage	452	3673
XMagickCommand_0	451	4970
ReadPCXImage	420	2716
ReadRLEImage	416	2832
GetImageFeatures	411	5799
ReadMIFFImage	411	2918
ReadPICTImage	411	2804
WritePNMImage	410	3139
XListBrowserWidget	402	2729
ReadRGBImage	400	3192
ReadBGRImage	398	3182
IdentifyImage	395	3830
ReadYCBCRImage	393	3156
WriteSVGImage	375	3178
ReadVIFFImage	364	2580
WritePSImage	350	3215
EncodeImageAttributes	343	3257
DistortImage	340	4452
ReadMPCImage	335	2397
MorphologyPrimitive	332	2008
WriteBMPImage	331	2600
XImageWindowCommand	316	1259

(a) Original

function_name	#nodes	instrnum
PreviewImage	494	3120
RaiseImage	492	2612
DrawPrimitive	492	2859
EvaluateImages	492	2458
ReadTIMImage	490	2787
XCommandWidget	489	2912
ReadPDBImage	487	2940
ReadPCDImage	485	2868
ReadFITSImage	483	2516
WriteMPCImage	480	3033
GetVirtualPixelsFromNexus	474	2844
XRotateImage	474	2596
ReadPSDImage	474	2725
XAnimateBackgroundImage	473	3244
InterpolatePixelChannel	469	2978
ReadXCFImage	465	2555
WriteMETAImage	462	2433
QueryColorCompliance	459	2477
SpliceImage	451	2450
ImportGrayQuantum	450	2600
EncodeImage_1	450	2348
WriteSGIImage	447	2443
SetImageAlphaChannel	444	2223
FloodfillPaintImage	444	2457

(b) Obfuscated with FLA

Figure 6.8: The number of basic blocks and the number of instructions of randomly selected functions in `ImageMagick` v7.0.6.10 `libMagickCore.7.Q16HDRI` library cross-compiled for x86 architecture with *clang* compiler

(e.g., compare and jump instructions). Moreover, obtaining VEX-IR from an assembly function is performed at basic block level. Besides, recent works [70, 224] consider basic blocks as the unit of their comparisons to find the similarities between programs, functions, or code fragments. Consequently, we consider each basic block as a word/token to be fed into our LSTM Encoder-Decoder model. The overall process of summarizing of a function and getting a new function representation is illustrated in Figure 6.9.

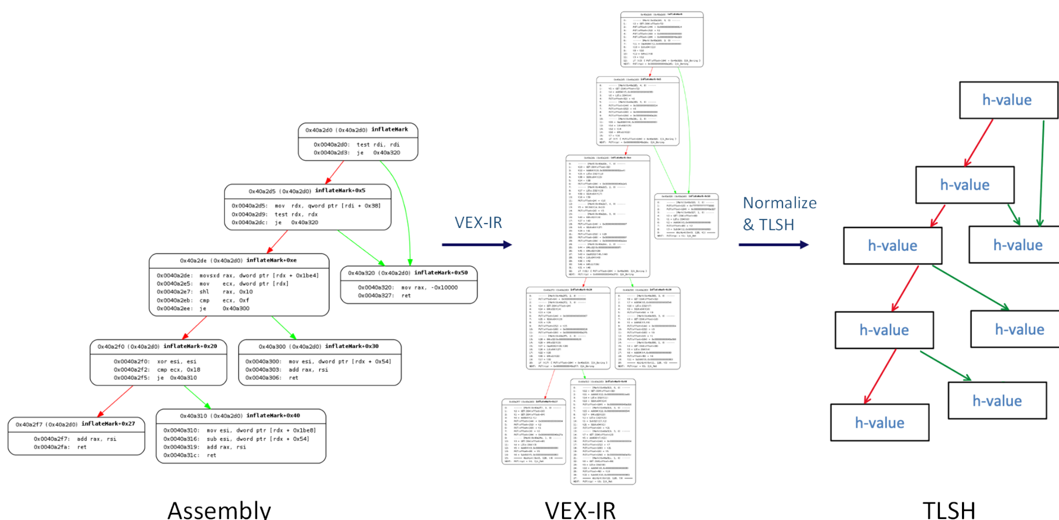


Figure 6.9: Function summarizing

Challenge 2: Vocabulary Size: The input to the LSTM encoder and decoder is in the form of word embeddings of the function representations. These word embeddings are learned during the training phase. One of the well-known challenges in the neural machine translation is Out-Of-Vocabulary (OOV) words, where these words have not been seen during the training phase, while they may appear in the test dataset. In these cases, since the model is not trained on the OOV words, it cannot represent those words. On the other hand, converting each basic block to a TLSH value significantly increases the size of our vocabulary. However, using a large vocabulary limits the applicability of the model due to computation or memory constraints [216], and some words in a very large vocabulary may contribute little to the translation process [46].

Solution to Challenge 2: To address this challenge, we first normalize VEX representations (e.g., memory addresses and register names) as explained in Section 6.4.1, which help decreasing

OOV words. We further reduce the size of vocabulary by following the same approach employed in natural language processing tasks [116, 147], where the vocabulary is limited to the top $50k$ most frequent words, and the words which are not part of this shortlist are generalized into `<unk>` token.

Reversed Source Sequences

The LSTM models are capable to learn long-term dependencies between words in long sentences. However, in [198] the authors demonstrate that reversing the source sentences in an LSTM Encoder-Decoder model will markedly increase the accuracy of the translation, especially on long sentences. Since reversing process results in introducing short-term dependencies. In fact, by reversing the source sentence, the first words in the source sentence will be close to the first words in the target sentence; this will reduce the problem's '*minimal time lag*' [106] and improve the performance [198]. Therefore, inspired by this work, we reverse the function representation sequences as the input to the LSTM encoder.

6.5.3 Function Matching

Given a binary code or firmware image acquired from an IoT device, we are interested in finding open-source library functions that are used in those binaries. As such, we perform code similarly detection against a large repository of known open-source libraries and their CVEs. More specifically, two binary functions are *similar*, if they originate from the same source code regardless of their underlying compilers (e.g., *GCC* and *clang*), optimization settings (e.g., `O0-O3`), and architectures (e.g., ARM and x86). Furthermore, possible CVEs are identified, if a given function is similar to a vulnerable function in our repository.

Given an unknown function to the trained LSTM Encoder-Decoder model, the model outputs the equivalent version of that function in another architecture. The *output* is provided with a sequence of TLSH tokens. We aim at computing the similarity between the output and the functions in our repository. To achieve this goal, we first represent each function and the translated function (*output*) with a numerical vector (embeddings) by employing the Distributed Memory Model of

Paragraph Vectors (PV-DM) model [136]. The PV-DM model is inspired by learning word embeddings methods, which utilizes the paragraph vectors to predict the next word of a paragraph. Moreover, the paragraph vectors take into consideration the semantics of the words as well as their order [136]. Second, to measure the similarity between the embeddings of a given *output* and the functions in our repository, the Triangle Similarity - Sector Similarity (TS-SS) similarity[5] [102] is utilized. This similarity score is recommended to be used for document comparisons [102]. Finally, top-k ranked similar functions from the repository will be returned as possible *candidate* functions.

6.6 Evaluation

In this section, we first provide the experimental setup and the evaluation metrics. Then, we compare TIOHTIÀ:KE with state-of-the-art approaches, and examine the effects of different code transformation techniques and hyper-parameters. Finally, we test TIOHTIÀ:KE on real-world firmware images to identify vulnerabilities.

6.6.1 Experimental Settings

We adapt the LSTM Encoder-Decoder model, which is implemented in Python using `keras` platform with `tensorFlow` as backend. Our experiments are conducted on a Linux server running Ubuntu 18.04 with 2x Intel Xeon Gold 5218 (16 cores and 2.30 GHz). We utilize an *SQLite*[6] database to store all the libraries and corresponding function embeddings. A specialized environment for reverse engineering and cross-compilation for the ARM and x86 architectures is created by using *Vagrant*[7]. The utilized compilers are *GCC* version 5.4.0 and *clang* version 4.0.1 using all compatible optimization flags, i.e., `O0-O3`. The symbol names are preserved during the compilation process for metric validation. A *Docker*[8] is used to create a containerized version of the

[5] https://github.com/taki0112/Vector_Similarity. Accessed on Dec 20, 2020.
[6] https://www.sqlite.org/index.html. Accessed on Dec 20, 2020.
[7] https://www.vagrantup.com/. Accessed on Dec 20, 2020.
[8] https://www.docker.com/. Accessed on Dec 20, 2020.

CVE database[9] and its associated search tools. The ANGR framework[10] [193] is utilized to setup the environment and lift the functions into the corresponding VEX representations. The Obfuscator-LLVM[11] [120] is employed to generate obfuscated versions of the functions in our repository using the following three flags: `fla`, `bcf`, and `sub`.

Datasets. In order to evaluate the proposed approach, we collect four different datasets as follows:

- **Dataset I:** This dataset includes two versions of the `OpenSSL` library, v1.0.1f and v1.0.1u, compiled with *GCC* v5.4 compiler and `O0-O3` optimization settings for ARM and x86 architectures. The dataset is built as in [151, 213].

- **Dataset II:** We build this dataset similar to [69], which consists of `Libgmp` v6.1.1, `Libcurl` v7.50.2, `ImageMagick` v7.0.6, `OpenSSL` v1.0.2s and `zlib` v1.2.7.1 libraries. We consider the binaries compiled with *GCC* compiler and `O0-O3` optimization settings.

- **Dataset III:** This dataset is composed of `libgmp` v6.1.0, `ImageMagic` v7.0.1, `OpenSSL` v1.0.1f, and `zlib` v1.2.7.1 libraries compiled with *clang* compiler and optimization `O2` with the three FLA, BCF and SUB obfuscation flags.

We build the aforementioned datasets, which consist of function pairs from two different architectures, i.e., ARM and x86. Each dataset is split into three disjoint subsets respectively for training, validation and test sets. We ensure that two binary functions originating from the same source code are not part of different subsets. For the evaluation purposes, we first train the model on the validation set and save the model. Then, we use the saved model with its corresponding hyper-parameters and evaluate the model on the test set.

6.6.2 Comparison

This section compares the accuracy results of our solution with the sate-of-the-art code similarity detection approaches (e.g., [69, 78, 84, 151, 213, 174]), which employ deep learning models, and

[9] `https://github.com/cve-search/cve-search`. Accessed on Dec 20, 2020.
[10] `https://github.com/angr/pyvex`, Accessed on Dec 20, 2020.
[11] `https://github.com/obfuscator-llvm/obfuscator/wiki`. Accessed on Dec 20, 2020.

identify vulnerabilities. Particularly, we compare TIOHTIÀ:KE with two best existing solutions, i.e., SAFE [151] and ASM2VEC [69] that support cross-architecture and code obfuscation. SAFE is reported to outperform existing cross-architecture code similarity detection [78, 84, 213, 174], and ASM2VEC is the only existing work that supports obfuscated binaries. According to our experiments, TIOHTIÀ:KE performs slightly better than ASM2VEC in handling obfuscation effects with the recall of 0.903 on average, while ASM2VEC's average recall is 0.849. Moreover, unlike ASM2VEC's single architecture support, TIOHTIÀ:KE can identify obfuscated binaries compiled for both x86 and ARM architectures. On the other hand, the accuracy of our approach (0.993) is similar as that of SAFE (0.992). Additionally, TIOHTIÀ:KE has the advantage of handling obfuscated binaries, whereas SAFE is not designed for this purpose. In the following, we elaborate on our comparative results.

Impact of Obfuscation

We conduct experiments to demonstrate the accuracy of our approach on obfuscated binaries and to compare it with ASM2VEC [69] approach. To this end, we prepare the same setup and similar dataset, i.e., *Dataset III*, as proposed in ASM2VEC. First, we train the model on an original library and test it against one of its obfuscated versions (e.g., FLA). Then, we train the model on the same obfuscated version and test it against its original binary. We follow the same process for other obfuscated libraries (e.g., BCF) and we report the average accuracy results for each set of experiments. Similar to ASM2VEC, we use precision and recall as our evaluation metrics. On the other hand, we preform the same experiments on *Dataset III* by deploying ASM2VEC[12]. The execution of ASM2VEC achieves lower precision results compared to our approach. Therefore, we present the results for the *Recall@1*.

As illustrated in Table 6.5, we achieve higher average recall of 0.849 for FLA obfuscation compared to ASM2VEC (0.699). For the other two obfuscation techniques, we are comparable since ASM2VEC supports only one architecture, while TIOHTIÀ:KE supports two architectures. The main

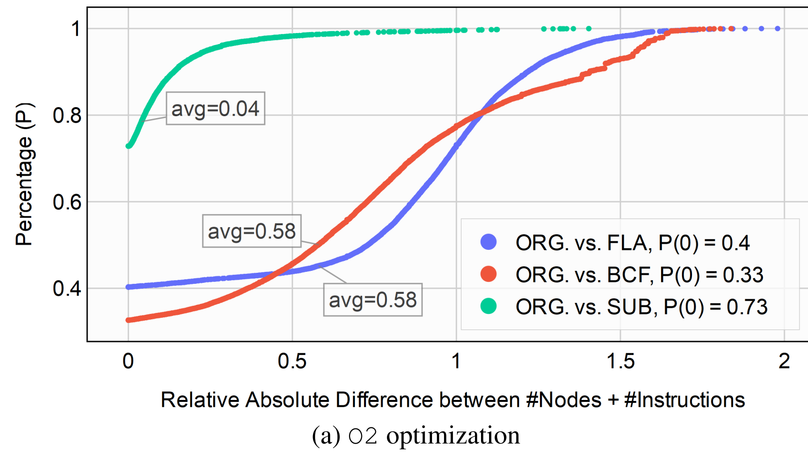
[12] https://github.com/McGill-DMaS/Kam1n0-Community. Accessed on Dec 20, 2020.

reason why TIOHTIÀ:KE performs similarly as ASM2VEC, in spite of its multiple architecture support (unlike ASM2VEC), is that TIOHTIÀ:KE learns function embeddings from the known function pairs originated from two different architectures, and translates one function representation into another representation of the same function.

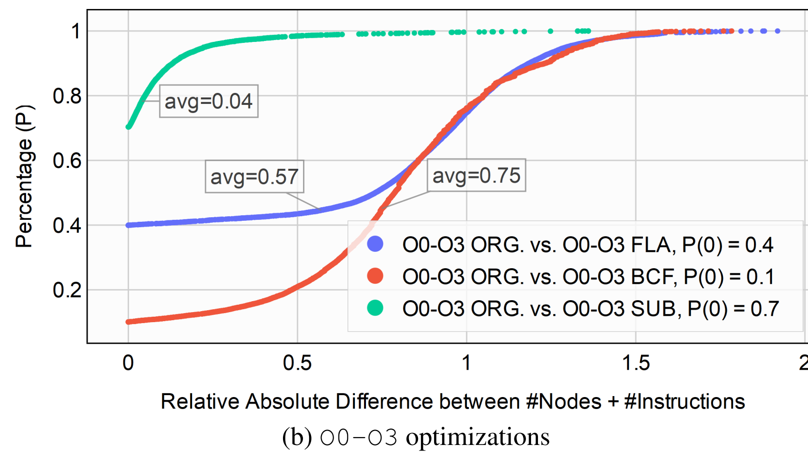
PROPOSAL	Obf.	libgmp	ImageMagick	openssl	zlib	Avg.
TIOHTIÀ:KE	FLA	0.861	0.827	0.867	0.842	0.849
(Cross-arch.)	BCF	0.923	0.878	0.971	0.904	0.919
	SUB	0.897	0.951	0.974	0.934	0.940
ASM2VEC	FLA	0.756	0.916	0.425	0.803	0.699
(Single-arch.)	BCF	0.960	0.912	0.880	0.900	0.913
	SUB	0.958	0.933	0.915	0.973	0.935

Table 6.5: Comparing the accuracy results of our cross-architecture (e.g., x86 and ARM) approach (TIOHTIÀ:KE) and state-of-the-art single-architecture (e.g., x86) approach (ASM2VEC) between the original and obfuscated binaries using *Recall@1*

We utilize Empirical Distribution Function (EDF) [38] to provide more insights about the differences between the original functions and their obfuscated versions. We calculate the differences between the number of nodes and the number of instructions on binaries compiled with the *clang* compiler and (i) optimization setting O2 (Figure 6.10a) and (ii) all the optimization settings together (Figure 6.10b). We observe that the largest difference between the original functions and their obfuscated versions (including all optimizations) is in the case of BCF obfuscation; 10% of the BCF obfuscated functions have the same number of nodes and instructions as their original versions, and the average differences is about 75% on the entire corpus. The BCF obfuscation has the largest effect on the combination of different optimization settings as well, while the effects of FLA and SUB techniques remain almost unchanged on average compared to considering only O2. This might be resulting from the fact that the BCF obfuscation techniques inject extra basic blocks in addition to the original functionality, which cause a much larger number of instructions and nodes in relation to the size of the original functions.



(a) `O2` optimization



(b) `O0-O3` optimizations

Figure 6.10: Empirical distribution between #nodes and #instructions in obfuscated binaries compiled with *clang* compiler

Compiler Optimizations

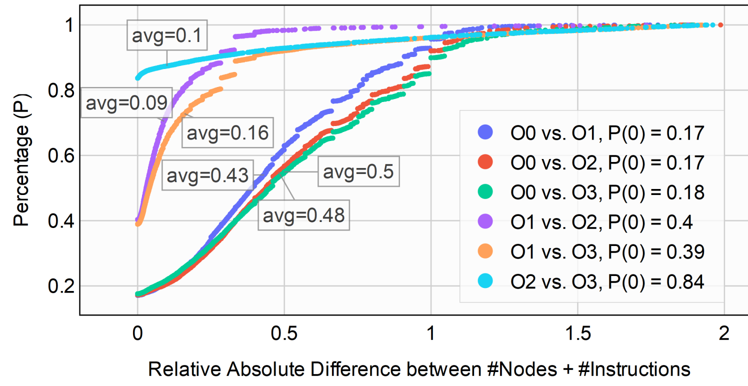
We evaluate the accuracy of our approach against different compiler optimizations and compare our results with those of ASM2VEC [69]. To this end, we first build the *Dataset II* and prepare the same setup similar to [69]. We train our model with a single library and one optimization level (e.g., O2) and test it against the same library and the next optimization level (e.g., O3). Conversely, we train the model with one optimization level (e.g., O3) for the same library and test it with the previous optimization level (e.g., O2). Finally, we provide the average of the obtained accuracy results. To compare our results with ASM2VEC, we report the best and worse scenarios as done in ASM2VEC. This includes the comparison of `O2` and `O3` optimizations (since the higher optimizations includes all the strategies from the lower levels) as well as the comparison between `O0` and `O3` optimizations.

We deploy ASM2VEC by considering the provided hyper-parameters and preform similar experiments using the same dataset. In these experiments, ASM2VEC shows significantly lower accuracy results than the reported results in the paper (which might have been caused by a mismatch in the configurations that are not explicitly mentioned in the paper or code repository). Therefore, we report the published results directly from the paper for comparison purposes in Table 6.6. As shown, TIOHTIÀ:KE and ASM2VEC perform almost the same, with a slight improvement in the case of the comparison between the `O0` and `O3` optimizations in favour of TIOHTIÀ:KE, considering the fact that TIOHTIÀ:KE supports two CPU architectures.

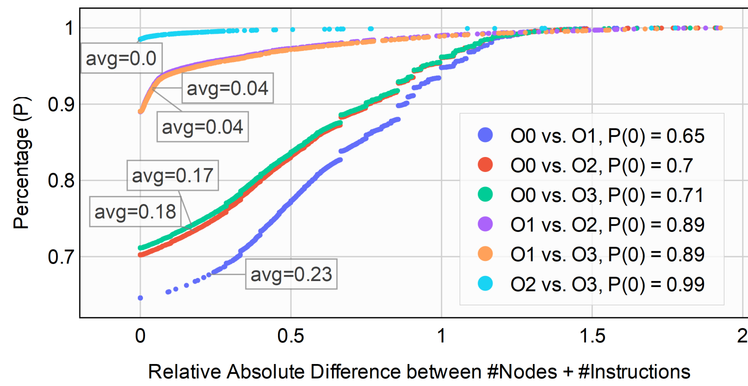
PROPOSAL	Opt.	libgmp	ImageMagick	libcur	openssl	zlib	Avg.
TIOHTIÀ:KE (Cross-arch.)	O2 vs. O3	0.932	0.971	0.943	0.975	0.861	0.936
	O0 vs. O3	0.801	0.731	0.770	0.789	0.867	0.848
ASM2VEC (Single-arch.)	O2 vs. O3	0.973	0.971	0.951	0.931	0.885	0.936
	O0 vs. O3	0.763	0.837	0.850	0.792	0.722	0.793

Table 6.6: Comparing the accuracy results of our cross-architecture (e.g., x86 and ARM) approach and reported results in the state-of-the-art single-architecture (e.g., x86) approach [69] (ASM2VEC) while varying the optimization settings for binaries compiled with *GCC* compiler using *Precision@1*

The pair-wise comparisons between each optimization setting and other optimization settings, in binaries compiled with the *GCC* and *clang* compilers, are presented in Figure 6.11. As for the binaries compiled with the *GCC* compiler, Figure 6.11a shows that the difference between `O0` and other optimization settings, i.e., `O1-O3`, is higher compared to other cases. As illustrated, the percentage of these function pairs to be similar is about $17\% - 18\%$, and about $43\% - 50\%$ of the function pairs differ from each other on average. In contrast, the difference between function pairs compiled with `O2` and `O3` has the lowest value; the percentage of the functions with the same number of nodes and instructions is 84%. On the other hand, the differences between the function pairs compiled with the *clang* compiler and `O1-O3` against `O0` optimization drops (Figure 6.11b), and the differences between the binaries compiled with `O2` and `O3` is nearly zero.



(a) *GCC* compiler



(b) *clang* compiler

Figure 6.11: Empirical distribution between #nodes and #instructions in binaries compiled with *GCC* abd *clang* compilers and all the optimization settings

Cross-architecture Binaries

To compare our solution with the state-of-the-art cross-architecture approache, we build *Dataset I* similar to [213, 151] approaches. Similar to SAFE [151], we perform 5-fold cross validation, in which we split the dataset into $five$ groups of approximately equal sizes, and for each group we generate the training and testing data. Then, we train the model on the training dataset and test it against the testing dataset. Finally, we report the average results of the independent experiments. In this settings, we measure the Receiver Operating Characteristic (ROC) performance measurement. We obtain the area under the ROC curve (AUC) of 0.993, which is very slightly better but comparable to the reported AUC of 0.992 in SAFE. In fact, the advantage of TIOHTIÀ:KE over SAFE is that TIOHTIÀ:KE can handle obfuscated binaries, while SAFE is not designed for that. Since the whole sequences of the instructions are fed into an RNN model proposed by SAFE, those

instruction sequences can be altered by the obfuscation techniques.

6.6.3 Accuracy Results

The primary objective of our experiments is to evaluate the accuracy of TIOHTIÀ:KE. We randomly select different libraries from our repository that are compiled with both *GCC* and *clang* compilers and O0−O3 optimization settings. Similar to [151], we perform 5-fold cross validation.

Impact of Compiler Optimization

In this section, we examine the effects of compiler optimizations on the accuracy of our approach. We first consider one library (e.g., zlib) compiled with the *GCC* compiler and all the optimization settings, i.e., O0−O3. Then, we perform 5-fold cross validation and report the average accuracy results using precision and recall performance metrics. We repeat the same experiments for the same library compiled with the *clang* compiler and all the optimization settings and report the average accuracy results. We carry out similar experiments for all the selected libraries. Obtained accuracy results are presented in Table 6.7 and Table 6.8.

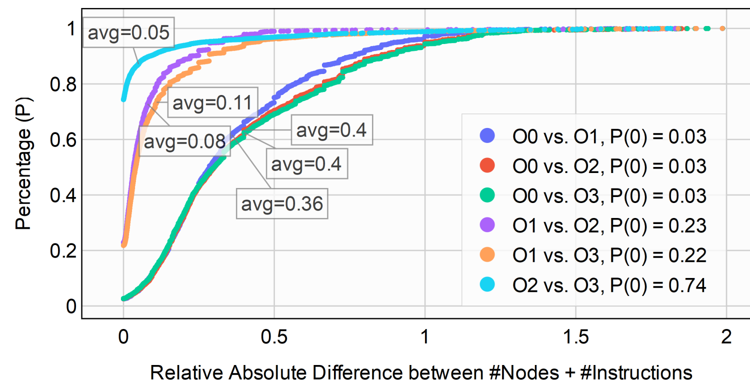
LIBRARY	Precision						
	Optimization (O0−O3)		Compiler (*GCC* & *clang*)				Average
	GCC	*clang*	O0	O1	O2	O3	
libgmp	0.895	0.913	0.911	0.860	0.901	0.901	0.890
Imagemagick	0.856	0.900	0.825	0.856	0.856	0.856	0.861
openssl	0.901	0.980	0.862	0.899	0.903	0.899	0.907
valgrind	0.873	0.899	0.801	0.825	0.833	0.841	0.849
zlib	0.905	0.901	0.825	0.859	0.860	0.862	0.869
All Libraries	Precision: 0.84 (k=1)						

Table 6.7: Accuracy results of TIOHTIÀ:KE using Precision

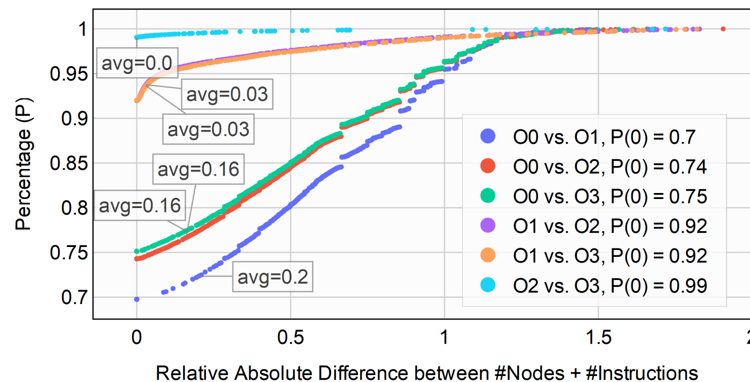
LIBRARY	Recall						
	Optimization (O0−O3)		Compiler (*GCC* & *clang*)				Average
	GCC	*clang*	O0	O1	O2	O3	
libgmp	0.814	0.975	0.882	0.841	0.882	0.893	0.879
Imagemagick	0.802	0.897	0.800	0.802	0.798	0.823	0.822
openssl	0.850	0.879	0.843	0.873	0.871	0.871	0.865
valgrind	0.827	0.816	0.780	0.804	0.811	0.819	0.810
zlib	0.874	0.860	0.798	0.834	0.848	0.839	0.842
All Libraries	Recall: 0.76 (k=20)						

Table 6.8: Accuracy results of TIOHTIÀ:KE using Recall

As illustrated in Table 6.7, we get the highest precision of 0.901 in identifying `OpenSSL` functions compiled with the *clang* compiler, and the lowest precision of 0.856 for the `ImageMagick` library compiled with the *GCC* compiler. To get additional insights, we perform statistical analysis by utilizing EDF to examine the differences between function pairs in `ImageMagick` as well as `OpenSSL` libraries. The results of these analyses are depicted in Figure 6.12a and Figure 6.12b, respectively. As shown, function pairs in `OpenSSL` library compiled with different optimization settings are more similar to each other (especially in the case of high optimization settings). For instance, 99% of function pairs compiled with `O2` and `O3` optimizations have the same number of nodes and instructions. On the other hand, 70% of function pairs compiled with the *GCC* compiler and `O2` and `O3` optimization in the `ImageMagick` library are similar. This will lead to higher accuracy results while identifying the functions in the `OpenSSL` library.



(a) `ImageMagick` compiled with *GCC*



(b) `OpenSSL` compiled with *clang* compiler

Figure 6.12: Empirical distribution between #nodes and #instructions in two libraries compiled with different compilers and optimization settings

Impact of Compilers

We further examine the effects of compilers by considering libraries that are compiled with two different compilers (*GCC* and *clang*) while fixing the optimization setting (e.g., O2). The obtained accuracy results are listed in Table 6.7. On average, `OpenSSL` library functions compiled with optimization `O2` are identified with the highest precision of 0.903, while identifying `libgmp` library functions with optimization `O3` has the highest recall of 0.893.

The effects of compilers, in terms of the number of nodes and instructions for different function pairs is illustrated in Figure 6.13. As shown, the more the code is optimized, the more similar the functions compiled with different compilers are. For instance, the distance between 31% of function pairs compiled with `O3` is zero and on average the distances are 0.16, while only 17% of them are similar when the code is least optimized (e.g., for `O0`).

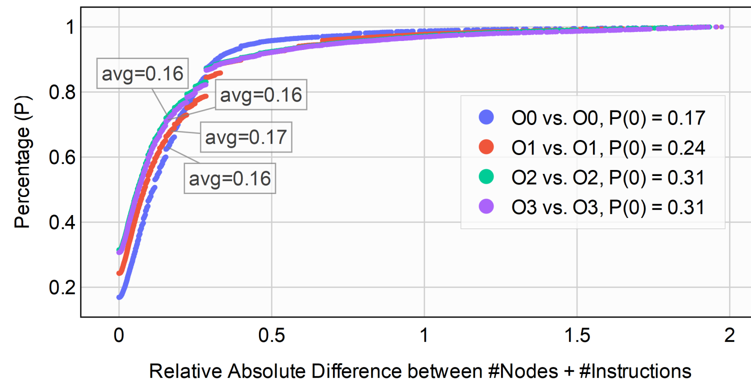


Figure 6.13: Empirical distribution between #nodes and #instructions in binaries compiled with *GCC* and *clang* compilers and the same optimization settings.

Accuracy Results for All Libraries

We repeat the experiments by considering all the libraries compiled with the *GCC* and *clang* compilers and all the optimization settings, i.e., `O0`−`O3`. The obtained results indicate the overall precision of 0.84 and the recall of 0.76 for the top-1 and top-20 (k=20) *candidate* functions, respectively.

As seen, the best results are obtained in the case where the compilers are fixed and the optimization settings vary (second and third sub-columns in Table 6.7). On the other hand, training the

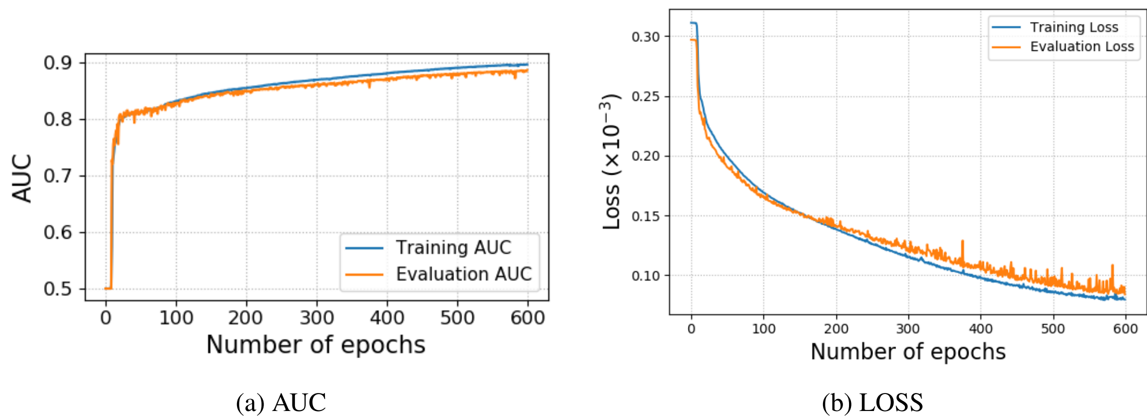
model with one compiler (e.g., *GCC*) and testing with the same library but compiled with another compiler (e.g., *clang*) gives the lowest accuracy results.

6.6.4 Hyper-parameters Selection

In this section, we investigate the effectiveness of hyper-parameters in our model. In particular, we consider *Dataset I* and examine the impact of the number of training epochs, embedding size, and the number of LSTM network layers.

Number of Epochs

We train our model with 600 epochs and evaluate the loss and AUC on the validation set. The obtained results are illustrated in Figure 6.14. As illustrated, after about 20 epochs the loss drops to a low value of 0.00023, and afterwards it keeps decreasing and gets closer to zero. The highest AUC value appears at epoch 600, which is equal to 0.989. Whilst the model can be trained for 20 epochs to achieve reasonably good AUC performance of 0.801, to get the best AUC it needs to be trained till 600 epochs.



(a) AUC

(b) LOSS

Figure 6.14: Effects of epoch on AUC and loss

Embedding size	AUC
150	0.752
200	0.811
250	0.890
300	0.891

(a) Embedding size

Latent dimension	AUC
250	0.890
300	0.969
350	0.977
400	0.895

(b) Latent dimension

# LSTM Layers	AUC
1	0.977
2	0.993
3	0.971

(c) # LSTM Layers

Table 6.9: Impact of different hyper-parameters

Embedding Size

In order to find the best optimum embedding size, we conduct separate experiments while independently varying word embedding size and latent dimension starting from 150 and 250, respectively. We increase these values by 50 each time, and measure their effects on the AUC, as listed in Table 6.9. As seen, selecting the values of 250 and 350 respectively for the embedding size and latent dimension, are the best choices. Although choosing the embedding size of 300 results in a slightly better accuracy, due to the trade-off between the accuracy and efficiency, we choose 250.

Network Layers

We further conduct experiments to examine the effects of the number of LSTM layers on the accuracy of our model. To this end, we fix the embedding size and latent dimension respectively to 250 and 350 (which are obtained from the previous experiments), and train the model by varying the number of LSTM layers over 600 epochs. The obtained results are illustrated in Table 6.9c. It can be seen that choosing two LSTM layers yields the highest AUC.

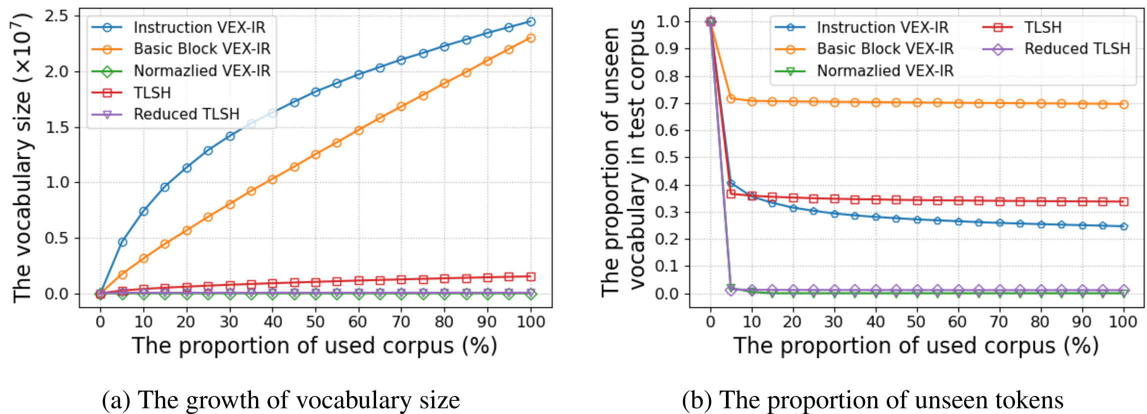
6.6.5 Out-Of-Vocabulary Words

In this section, we evaluate the effect of out-of-vocabulary (OOV) words. We consider the entire corpus in order to perform two sets of experiments as follows.

The Growth of Vocabulary Size. We examine how much the size of the vocabulary grows (i) before performing any processing on the VEX presentation, (ii) after normalization, and finally

(iii) when we apply the TLSH on the normalized basic blocks. To this end, we consider all the words/tokens in the whole corpus excluding two randomly selected binaries, which are reserved for the next set of experiments. Then, we divide the corpus equally into 20 parts. For each part, we count the vocabulary size in terms of the corpus percentage that has been already analyzed. The obtained results are presented in Figure 6.15a. As seen, before the normalization process (e.g., Instruction VEX-IR and Basic Block VEX-IR), the more the proportion of the corpus is used, the larger the vocabulary size is. Whilst, the vocabulary size remains almost the same within the whole corpus in the case of using the TLSH values similar to the normalized VEX-IR.

We additionally count the vocabulary size per basic block without applying the TLSH, i.e., Basic Block VEX-IR. As shown, the vocabulary size increases with the growth of the used corpus. These results confirm the effectiveness of our preprocessing and TLSH generation solutions.



(a) The growth of vocabulary size

(b) The proportion of unseen tokens

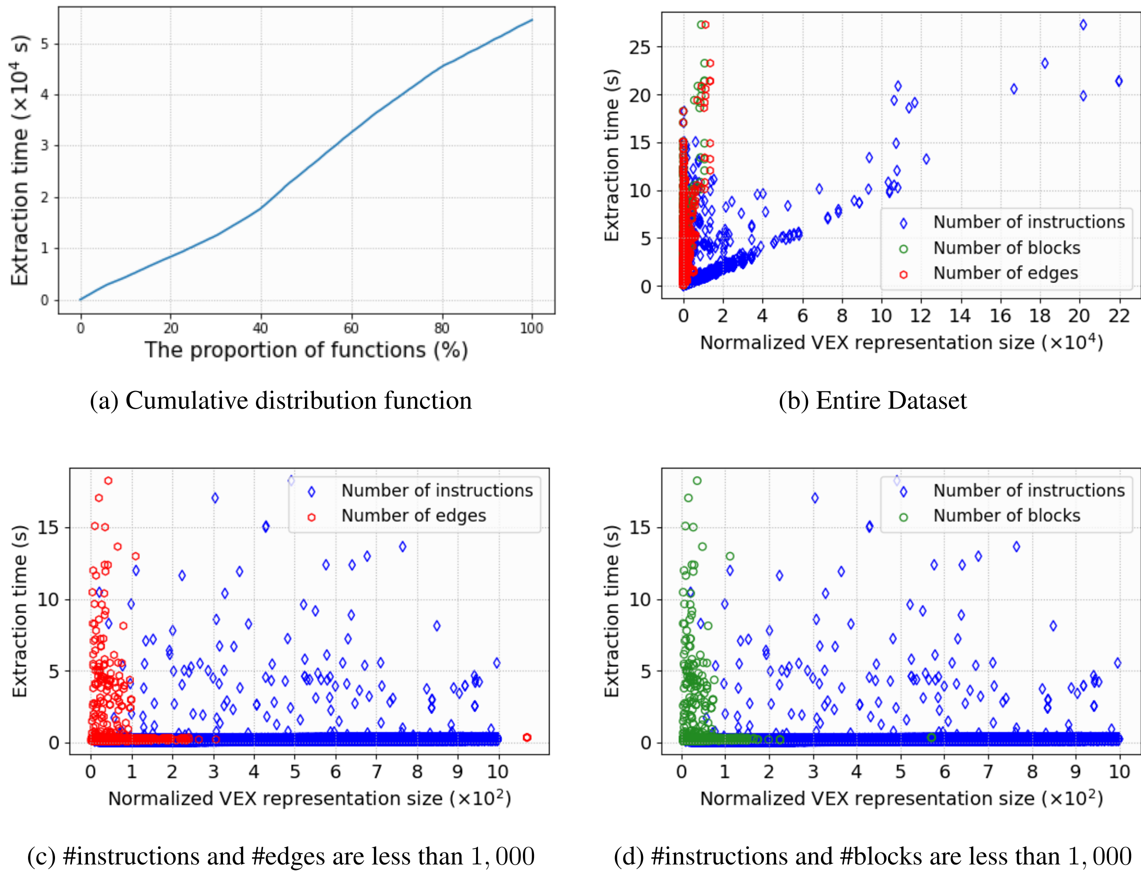
Figure 6.15: Evaluation on out-of-vocabulary tokens

The Proportion of Unseen Tokens. We further select two binaries that were excluded from the corpus of the previous experiments. Then, we count the percentage of unseen tokens that do not exist in the obtained vocabulary from the previous experiments. We perform this experiments at both VEX-IR instruction and basic block levels, normalized VEX-IR and TLSH values. The obtained results are presented in Figure 6.15b , which indicate that the proportion of unseen tokens in the VEX representation is more than that of TLSH tokens.

6.6.6 Efficiency

In this section, we evaluate the efficiency of our approach with respect to normalized VEX-IR generation and training time. We consider *Dataset II*, which contains five libraries and $261,570$ function pairs for the x86 and ARM architectures.

Normalized VEX-IR Extraction Time. We measure the efficiency of our approach to extract the normalized VEX-IR, which includes the process of lifting the disassembled functions into the VEX representation along with the normalization process and indexing. In this setting, we use multi-threading in order to enhance the efficiency.



(a) Cumulative distribution function

(b) Entire Dataset

(c) #instructions and #edges are less than $1,000$

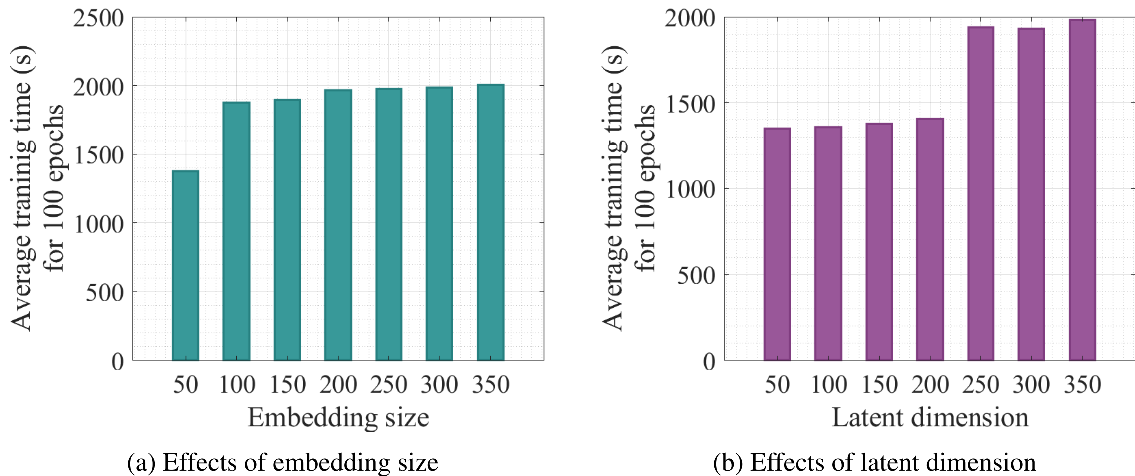
(d) #instructions and #blocks are less than $1,000$

Figure 6.16: Efficiency evaluation on Dataset II

We measure the cumulative distribution function (CDF) to extract normalized VEX-IR functions as presented in Figure 6.16a. Furthermore, the extraction time for the normalized VEX-IR

with respect to functions size is illustrated in Figure 6.16b. For the sake of clarity, a subset of the obtained results for functions with less than $1,000$ instructions, edges and blocks are represented in Figure 6.16c and Figure 6.16d, respectively. We observe that the extracting time increases the most when the number of edges and the number of blocks increase compared to the increase in the size of instructions. We also measure the average extraction time (excluding the indexing time) per library, and we obtain 0.169, 0.125, 0.140, 0.101 and 0.131 seconds for the `ImageMagick`, `libgmp`, `libcurl`, `OpenSSL`, and `zlib` libraries, respectively. The average indexing time per function in all the libraries, on the other hand, is 0.0603 seconds.

Training Time. We further examine the effects of different hyper-parameters on the training time. First, we train the model by varying the embedding size starting from 50 to 350 over 100 epochs. The average training time is illustrated in Figure 6.17a. Then, we measure the effects of latent dimension on training time by fixing the embedding size to 100 and varying the latent dimension. As shown in Figure 6.17, the higher the embedding size or latent dimension is, the slower the training of the model is. However, since the training process is a one-time task, in our evaluations we consider for each parameter the value that gives the best comparable accuracy regardless of their training time. Moreover, definitely using GPUs will speed up the training time significantly.



(a) Effects of embedding size

(b) Effects of latent dimension

Figure 6.17: Training time with varying embedding size and latent dimension

Furthermore, we measure the training time of our model by fixing the embedding size at 30,

and varying the number of LSTM layers from one to three. We change the LSTM layers uniformly for both encoder and decoder. For each setting, we also change the size of the latent dimension to 30 and 50. The training time with respect to different LSTM layers is illustrated in Figure 6.18.

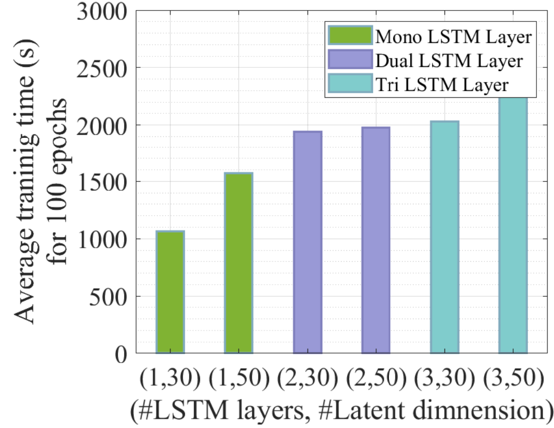


Figure 6.18: Training time with varying embedding size and number of network layers

6.6.7 Detecting Vulnerabilities on Real-World Firmware

ReadyNAS v6.1.6		*NI PMU1_0_11*		*Honewell.RTUR150*	
CVE	**Similarity**	**CVE**	**Similarity**	**CVE**	**Similarity**
CVE-2010-1633	0.865	CVE-2016-6303	0.966	CVE-2016-0701	0.983
CVE-2014-0160	0.863	CVE-2014-8176	0.971	CVE-2016-2105	0.963
CVE-2015-0288	0.900	CVE-2015-0288	0.899	CVE-2010-1633	0.915
CVE-2014-3566	0.750	CVE-2010-1633	0.881	CVE-2016-6303	0.897

Table 6.10: Identifying CVEs in real-world firmware images

In this section, we demonstrate the capability of TIOHTIÀ:KE to enable the vulnerability identification in real-world IED firmware. As such, we choose three publicly available firmware images from our firmware database. We choose the *Netgear ReadyNAS v6.1.6*[13] firmware image, which is available for both ARM and x86 architectures. We train our model on *Dataset I*, which consists of different versions on the `OpenSSL` library. The inputs to the encoder and decoder are the function pairs, <ARM, x86>, cross-compiled for the ARM and x86 architectures, respectively. Then, we

[13] http://www.downloads.netgear.com/files/GDC/READYNAS-100/ReadyNASOS-6.1.6-arm.zip. Accessed on Dec 20, 2020.

test our model with *Netgear ReadyNAS v6.1.6* firmware image for x86 architecture. Each resulting function pair is ranked using the similarity score. We consider a *candidate* as a potential match, if the similarity score is higher than 75%. A subset of obtained results is illustrated in Table 6.10. As seen, TIOHTIÀ:KE can identify the Heartbleed vulnerability (CVE-2014-0160), which is a serious vulnerability in the TLS and DTLS implementations of `OpenSSL` v1.0.1 (before v1.0.1g), in the *Netgear ReadyNAS v6.1.6* firmware image. This is also demonstrated by a number of state-of-the-art approaches [78, 84, 189].

We further consider the *PMU1_0_11*[14] and *Honeywell.RTUR150* firmware images, which are based on the ARM architecture. In this setting, we train our model on *Dataset I*, where the inputs to the encoder and decoder are the function pairs in `<x86, ARM>` architectures, respectively. Then, we test our model with the *PMU1_0_11* firmware image based on the ARM architecture. We can identify the CVE-2015-0288 in the *NI PMU1_0_11* firmware image. This vulnerability is identified by finding a matche with the `X509_to_X509_REQ` function in `OpenSSL` in our repository. We perform similar test for the *Honeywell.RTUR150* firmware image. The obtained results confirm the capability of TIOHTIÀ:KE for performing vulnerable function search on the real-world firmware images. Moreover, it demonstrates the TIOHTIÀ:KE's capability to match known vulnerable functions in binaries compiled for one architecture (e.g., x86) to unknown functions compiled for another architecture (e.g., ARM) and vice versa.

6.7 Limitations and Concluding Remarks

In this chapter, we proposed a cross-architecture code similarity detection approach with an application to vulnerability detection that also supports obfuscated binaries. To this end, we first built a large-scale vulnerability database that consists of most relevant and common cross-architecture cross-compiled vulnerable open-source libraries, that are used in the smart grid context. Then, we lifted the disassembeld functions into an intermediate representation and modeled function representations with a sequence of execution paths. Afterwards, we utilized an LSTM Encoder-Decoder

[14] `http://digital.ni.com/public.nsf/allkb/5391E8424944D0BC86257E45000B025C`. Accessed on Jan 15, 2018.

architecture to translate functions between different CPU architectures (e.g., ARM to x86). Next, we perform function matching for code similarity and vulnerability detection. We further applied our approach to real-world firmware images and identified potential CVEs.

However, TIOHTIÀ:KE has currently the following limitations, which will potentially be addressed in our future works. First, it currently does not consider the effects of function inlining. Also, our current training phase might be more efficient through performing feature selection prior to learning. Additionally, we have not considered the effects of other obfuscators (e.g., Tigress [22]). Finally, we intend to evaluate our approach for other architectures (e.g., MIPS).

Chapter 7

Conclusion

The popularity of digital technologies, especially with the development of Internet of Things (IoT), has seen a significant growth. Information technology (IT) and IoT devices are almost everywhere, ranging from consumer electronics and home networks to industrial sectors, such as healthcare, transportation, industrial control systems, retail, and smart cities. However, this growing popularity of IT turns it into a major subject of cybersecurity threats. On the other hand, the number of vulnerabilities are increasing[1]. Therefore, there is an urgent need of techniques that allow for code security assessment to uncover vulnerabilities. To address this gap, this thesis provides automated solutions for fingerprinting binary code and identifying known security vulnerabilities in program executables and firmware images of IoT devices. More specifically, we addressed several threats of research on binary code fingerprinting that targeted several applications, mainly, compiler provenance attribution, library function identification, code similarity detection and vulnerable function detection in cross-architecture and cross-compiler obfuscated binaries.

More specifically, we first presented a technique called BinComp for recovering compiler provenance of program binaries using syntactic, semantic, and structural features to capture compiler behaviors. BinComp can provide information about the build environment and also label compiler-related functions, the latter will accelerate binary analysis tasks. Second, we introduced

[1] https://nvd.nist.gov/general/visualizations/vulnerability-visualizations/cvss-severity-distribution-over-time. Accessed on Jan 21, 2021.

BINSHAPE with novel concept to representing a function based on its shape and proposed accordingly a robust signature for each standard library function from diverse collection of heterogeneous features. In addition, we designed a novel data structure to efficiently support accurate and scalable detection. BINSHAPE labels standard library functions, which accelerates binary analysis tasks and contributes to the improvement of accuracy. Third, we built a multi-stage detection engine called BINARM to efficiently identify vulnerable functions in IoT (e.g., intelligent electronic devices in the smart grid) firmware images, while maintaining high accuracy. BINARM can detect free-open-source library functions in firmware images and eventually identify possible security vulnerabilities that exist in those identified functions in the ARM architecture. Identifying vulnerable functions contributes towards code assessment and patch analysis in normal binaries and IoT devices. Finally, we leveraged neural machine translation techniques for code similarity and vulnerable function detection in cross-architecture cross-compiled obfuscated binaries and firmware images. This capability will help detect reused free open-source libraries and vulnerable functions in a binary code compiled with different compilers for various architectures in the presence of obfuscation.

To summarize, our four contributions together can help to label different types of functions in a given binary, namely compiler functions, standard library functions, free open-source library functions and vulnerable functions. Identifying the first three types of the functions will assist security analyst to first gain information about the underlying functionalities of that binary (e.g., file sharing), and then shift focus to unknown functions for more accurate and efficient analysis. Vulnerable function detection can help to assess the security of a binary or firmware images, for instance to ascertain that a provided patch does not contain known security vulnerabilities.

Our work can be extended in several directions. First, our methods are not currently designed to identify function inlining. In the future, we will consider data flow analysis and symbolic execution as potential solutions to this problem. Second, our proposed feature extraction and detection approaches along with statistical analysis can be applied on binaries compiled for different CPU architectures (e.g., ARM and x86) in order to provide comprehensive insights on the effects of different architectures (e.g., CISC[2] versus RISC[3]) at binary level. For instance, ARM instructions

[2]Complex Instruction Set Computing
[3]Reduced Instruction Set Computing

operate only on registers with a few instructions for reading/writing data from/to memory, while x86 can operate directly on memory. We aim at capturing these subtle but important effects on features imposed by different architectures and obfuscation techniques. Third, by taking further advantage from the set of high-value features introduced in this thesis, we might enhance the detection models based on advanced machine learning techniques and deep neural networks. Fourth, our proposed techniques can be improved to support other obfuscation techniques, such as virtualization and jilting, by utilizing a set of dynamic features. Furthermore, enhanced machine learning techniques that include adversarial models can be further investigated. Finally, our current approaches extensively used static analysis, which can be enhanced by exploring dynamic analysis or a hybrid technique (combination of both static and dynamic analyses).

In summary, this thesis shows the efficacy of machine learning in conjunction with conventional binary static analysis to solve various binary fingerprinting problems. Furthermore, this work shows that natural language processing in binary code analysis can contribute to extract relevant code semantics, which could not be easily extracted through other traditional static approaches. Additionally, this thesis explored on understanding the inner-working of obfuscation (e.g., control flow flattening and bogus control flow graph). Moreover, the proposed methods in this thesis can potentially be integrated with existing dynamic approaches to further propose a more in-depth binary analysis solution. Finally, our binary analysis can be complemented with other types of analysis, such as network analysis, to provide even more comprehensive solutions to defend against various cyber-threats in today's digital infrastructures.

Contributions of Authors

In what follows, a description of authors' contributions on each of the proposed works is provided.

Survey on Static Binary Analysis Approaches. In Chapter 2, we provide a comprehensive review along with quantitative and qualitative comparisons of state-of-the-art function matching and vulnerability detection approaches. My contribution is essentially to review static analysis and code similarity detection solutions, and to propose layered taxonomies for both approaches and utilized features, followed by open challenges and lesson learned. This work has been conducted with Abdullah Qasem, who performed a complementary survey of dynamic analysis approaches as well as collaborators from Hydro-Québec and Thales, who provided valuable comments and feedback. This survey has been accepted to be published in ACM Computing Surveys (CSUR).

Compiler Provenance Attribution. In Chapter 3, we devise a practical approach for compiler provenance attribution. Our main contribution is to propose a multi-layered approach, which analyzes the syntax, structure, and semantics of functions in order to identify the compilers, compiler functions, and compiler versions and optimizations at each layer. My main contribution is in the second layer in which we label compiler-related functions and also the implementation and evaluation of the state-of-the-art approaches. This work was a collaboration with Ashkan Rahimian and Saed Alrabaee, who contributed in two other layers of the approach and the implementation.

Library Function Identification. In Chapter 4, we propose a scalable and robust system to identify standard library functions in binaries. My main contribution is to propose a library function identification approach that derives a robust signature for each library function based on heterogeneous features and incorporates a novel data structure to store such signatures and facilitate efficient matching against a target function.

Vulnerability Detection in Firmware Images. In Chapter 5, we propose a scalable and accurate solution to detect vulnerable functions in firmware images. My main contribution is to propose a multi-stage detection mechanism, where we first filter out the irrelevant functions through the inexpensive operations, and then perform comparatively expensive and more accurate operations on smaller set of functions to detect the final vulnerable functions. This work has been conducted

with Leo Collard, who contributed in developing the idea of one of the stages and implementation of the approach, as well as collaborators from Hydro-Québec and Thales, who provided valuable comments and feedback.

Cross-Architecture Code Similarity and Vulnerability Detection. In Chapter 6, we propose a code similarity detection approach to identify vulnerable functions in binaries compiled with different compilers for different architectures in the presence of obfuscation techniques. My main contribution is to identify obfuscation-resilient feature and propose a neural machine translation-based solution, where we leverage the similarity between the translation of natural language processing (e.g., English to French) and code similarity detection for assembly language resulted from different CPU architectures (e.g., ARM and x86). This work has been conducted with Ling Tan, who mainly contributed in the implementation of the approach, as well as the collaborators from Hydro-Québec and Thales, who provided valuable comments and feedback.

Bibliography

[1] P. Agrawal, A. Arasu, and R. Kaushik. On indexing error-tolerant set containment. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 927–938. ACM, 2010.

[2] M. Ahmadvand, A. Pretschner, and F. Kelbert. A taxonomy of software integrity protection techniques. In *Advances in Computers*, volume 112, pages 413–486. Elsevier, 2019.

[3] S. Alrabaee. *Efficient, scalable, and accurate program fingerprinting in binary code*. PhD thesis, Concordia University, 2018.

[4] S. Alrabaee, M. Debbabi, P. Shirani, L. Wang, A. Youssef, A. Rahimian, L. Nouh, D. Mouheb, H. Huang, and A. Hanna. *Binary code fingerprinting for cybersecurity: Application to malicious code fingerprinting*, volume 78. Springer Nature, 2020.

[5] S. Alrabaee, M. Debbabi, and L. Wang. On the feasibility of binary authorship characterization. *Digital Investigation*, 28:S3–S11, 2019.

[6] S. Alrabaee, M. Debbabi, and L. Wang. CPA: Accurate cross-platform binary authorship characterization using LDA. *IEEE Transactions on Information Forensics and Security (TIFS)*, 15:3051–3066, 2020.

[7] S. Alrabaee, E. B. Karbab, L. Wang, and M. Debbabi. BinEye: Towards efficient binary authorship characterization using deep learning. In *European Symposium on Research in Computer Security (ESORICS)*, pages 47–67. Springer, 2019.

[8] S. Alrabaee, N. Saleem, S. Preda, L. Wang, and M. Debbabi. OBA2: An onion approach to binary code authorship attribution. *Digital Investigation*, 11:S94–S103, 2014.

[9] S. Alrabaee, P. Shirani, M. Debbabi, and L. Wang. On the feasibility of malware authorship attribution. In *International Symposium on Foundations and Practice of Security (FPS)*, pages 256–272. Springer, 2016.

[10] S. Alrabaee, P. Shirani, L. Wang, and M. Debbabi. SIGMA: A semantic integrated graph matching approach for identifying reused functions in binary code. *Digital Investigation*, 12:S61–S71, 2015.

[11] S. Alrabaee, P. Shirani, L. Wang, and M. Debbabi. FOSSIL: a resilient and efficient system for identifying FOSS functions in malware binaries. *ACM Transactions on Privacy and Security (TOPS)*, 21(2):1–34, 2018.

[12] S. Alrabaee, P. Shirani, L. Wang, M. Debbabi, and A. Hanna. On leveraging coding habits for effective binary authorship attribution. In *European Symposium on Research in Computer Security (ESORICS)*, pages 26–47. Springer, 2018.

[13] S. Alrabaee, P. Shirani, L. Wang, M. Debbabi, and A. Hanna. Decoupling coding habits from functionality for effective binary authorship attribution. *Journal of Computer Security*, 27(6):613–648, 2019.

[14] S. Alrabaee, L. Wang, and M. Debbabi. BinGold: Towards robust binary analysis by extracting the semantics of binary code as semantic flow graphs (SFGs). *Digital Investigation*, 18:S11–S22, 2016.

[15] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pages 459–468. IEEE, 2006.

[16] R. Arandjelovic and A. Zisserman. All about VLAD. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1578–1585, 2013.

[17] D. Babić, L. Martignoni, S. McCamant, and D. Song. Statically-directed dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA)*, pages 12–22. ACM, 2011.

[18] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *International conference on Compiler Construction (CC)*, pages 5–23. Springer, 2004.

[19] G. Balakrishnan and T. Reps. WYSINWYX: What you see is not what you eXecute. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(6):1–84, 2010.

[20] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum. WYSINWYX: What You See Is Not What You eXecute. In *Verified Software: Theories, Tools, Experiments (VSTTE)*, pages 202–213. Springer, 2008.

[21] M. Balliu, M. Dam, and R. Guanciale. Automating information flow analysis of low level code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1080–1091. ACM, 2014.

[22] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Computer Security Applications Conference (ACSAC)*, pages 189–200, 2016.

[23] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley. BYTEWEIGHT: Learning to recognize functions in binary code. In *23rd USENIX Security Symposium*, pages 845–860, 2014.

[24] S. Bardin, P. Herrmann, J. Leroux, O. Ly, R. Tabary, and A. Vincent. The BINCOA framework for binary code analysis. In *International Conference on Computer Aided Verification (CAV)*, pages 165–170. Springer, 2011.

[25] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *International Symposium on Formal Methods for Components and Objects (FMCO)*, pages 364–387. Springer, 2005.

[26] C. Barrett, D. Kroening, and T. Melham. Problem solving for the 21st century: Efficient solver for satisfiability modulo theories. 2014.

[27] F. Bellard. QEMU, a fast and portable dynamic translator. In *USENIX Annual Technical Conference (ATC), FREENIX Track*, volume 41, page 46, 2005.

[28] D. Bilar. Opcodes as predictor for malware. *International Journal of Electronic Security and Digital Forensics*, 1(2):156–168, 2007.

[29] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[30] M. Bourquin, A. King, and E. Robbins. BinSlayer: Accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW)*, pages 1–10. ACM, 2013.

[31] J. Bromley, I. Guyon, Y. LeCun, E. Säckinger, and R. Shah. Signature verification using a "siamese" time delay neural network. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 737–744, 1994.

[32] M. B. Brown and W. J. Dixon. *BMDP statistical software*. University of California Press, 1983.

[33] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A binary analysis platform. In *International Conference on Computer Aided Verification (CAV)*, pages 463–469. Springer, 2011.

[34] D. Brumley, I. Jager, E. J. Schwartz, and S. Whitman. The BAP handbook, 2013.

[35] J. Caballero and Z. Lin. Type inference on executables. *ACM Computing Surveys (CSUR)*, 48(4):1–35, 2016.

[36] C. Cadar, D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation (OSDI)*, pages 209–224. USENIX Association, 2008.

[37] A. Caliskan-Islam, F. Yamaguchi, E. Dauber, R. Harang, K. Rieck, R. Greenstadt, and A. Narayanan. When coding style survives compilation: De-anonymizing programmers from executable binaries. *The 25th Annual Network and Distributed System Security Symposium (NDSS)*, pages 255–270, 2018.

[38] R. Castro. The empirical distribution function and the histogram. *Lecture Notes, 2WS17-Advanced Statistics. Department of Mathematics, Eindhoven University of Technology*, 4, 2015.

[39] S.-H. Cha. Comprehensive survey on distance/similarity measures between probability density functions. *City*, 1(2):1, 2007.

[40] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley. Unleashing mayhem on binary code. In *IEEE Symposium on Security and Privacy (S&P)*, pages 380–394. IEEE, 2012.

[41] 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 (FSE)*, pages 678–689. ACM, 2016.

[42] T. E. Cheatham, G. H. Holloway, and J. A. Townley. Symbolic evaluation and the analysis of programs. *IEEE Transactions on Software Engineering (TSE)*, (4):402–417, 1979.

[43] B. Chen, X. Dong, G. Bai, S. Jauhar, and Y. Cheng. Secure and efficient software-based attestation for industrial control devices with arm processors. In *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC)*, pages 425–436, 2017.

[44] D. D. Chen, M. Egele, M. Woo, and D. Brumley. Towards automated dynamic analysis for linux-based embedded firmware. In *The Network and Distributed System Security Symposium (NDSS)*, pages 1–16, 2016.

[45] K. Chen, P. Liu, and Y. Zhang. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *Proceedings of the 36th International Conference on Software Engineering (ICSE)*, pages 175–186. ACM, 2014.

[46] W. Chen, Y. Su, Y. Shen, Z. Chen, X. Yan, and W. Wang. How large a vocabulary does text classification need? a variational approach to vocabulary selection. *arXiv preprint arXiv:1902.10339*, 2019.

[47] L. Cheng, K. Tian, and D. D. Yao. Orpheus: Enforcing cyber-physical execution semantics to defend against data-oriented attacks. pages 315–326, 2017.

[48] A. Cherepanov. WIN32/INDUSTROYER: A new threat for industrial control systems. *White paper, ESET (June 2017)*, 2017.

[49] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[50] C. Cifuentes and A. Fraboulet. Intraprocedural static slicing of binary executables. In *Proceedings International Conference on Software Maintenance (ICSM)*, pages 188–195. IEEE, 1997.

[51] L. Collard. Fingerprinting vulnerabilities in intelligent electronic device firmware. Master's thesis, Concordia University, 2018.

[52] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[53] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press, 2009.

[54] A. Costin, J. Zaddach, A. Francillon, D. Balzarotti, and S. Antipolis. A large-scale analysis of the security of embedded firmwares. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 95–110, 2014.

[55] M. Cova, V. Felmetsger, G. Banks, and G. Vigna. Static detection of vulnerabilities in x86 executables. In *2006 22nd Annual Computer Security Applications Conference (ACSAC)*, pages 269–278. IEEE, 2006.

[56] P. D. Coward. Symbolic execution systems- a review. *Software Engineering Journal*, 3(6):229–239, 1988.

[57] A. Cui, M. Costello, and S. J. Stolfo. When firmware modifications attack: A case study of embedded exploitation. In *the Network and Distributed System Security Symposium (NDSS)*, 2013.

[58] P. Cunningham and S. J. Delany. k-nearest neighbour classifiers. *Multiple Classifier Systems*, 34(8):1–17, 2007.

[59] H. Dai, B. Dai, and L. Song. Discriminative embeddings of latent variable models for structured data. In *International Conference on Machine Learning (ICML)*, pages 2702–2711, 2016.

[60] Y. David, N. Partush, and E. Yahav. Statistical similarity of binaries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 266–280. ACM, 2016.

[61] Y. David, N. Partush, and E. Yahav. Similarity of binaries through re-optimization. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 79–94. ACM, 2017.

[62] Y. David, N. Partush, and E. Yahav. Firmup: Precise static detection of common vulnerabilities in firmware. *ACM SIGPLAN Notices*, 53(2):392–404, 2018.

[63] Y. David and E. Yahav. Tracelet-based code search in executables. *ACM SIGPLAN Notices*, 49(6):349–360, 2014.

[64] D. Davidson, B. Moench, T. Ristenpart, and S. Jha. FIE on firmware: Finding vulnerabilities in embedded systems using symbolic execution. In *22nd USENIX Security Symposium (USENIX Security 13)*, pages 463–478, 2013.

[65] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340. Springer, 2008.

[66] E. Dimitriadou, S. Dolničar, and A. Weingessel. An examination of indexes for determining the number of clusters in binary data sets. *Psychometrika*, 67(1):137–159, 2002.

[67] A. Dinaburg and A. Ruef. McSema: Static translation of x86 instructions to LLVM. In *REcon 2014 Conference, Montreal, Canada*, 2014.

[68] S. H. Ding, B. 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 (KDD)*, pages 461–470. ACM, 2016.

[69] 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 *Proceedings of the 40th International Symposium on Security and Privacy (S&P)*, pages 38–55, San Francisco, CA, May 2019. IEEE Computer Society.

[70] Y. Duan, X. Li, J. Wang, and H. Yin. DeepBinDiff: Learning program-wide code representations for binary diffing. In *Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS)*, 2020.

[71] T. Dullien and S. Porst. REIL: A platform-independent intermediate representation of disassembled code for static code analysis, 2009.

[72] T. Dullien and R. Rolles. Graph-based comparison of executable objects (english version). *Symposium on Information and Communications Security (SSTIC)*, 5:1–3, 2005.

[73] C. Eagle. *The IDA pro book: the unofficial guide to the world's most popular disassembler*. No Starch Press, 2011.

[74] M. Egele, T. Scholte, E. Kirda, and C. Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)*, 44(2):6, 2012.

[75] 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)*, pages 303–317, 2014.

[76] A. Ehrenfeucht. An application of games to the completeness problem for formalized theories. *Fundamenta Mathematicae*, 49(129-141):13, 1961.

[77] K. L. Elmore and M. B. Richman. Euclidean distance as a similarity metric for principal component analysis. *Monthly Weather Review*, 129(3):540–549, 2001.

[78] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla. discovRE: Efficient cross-architecture identification of bugs in binary code. In *Proceedings of the 23rd Symposium on Network and Distributed System Security (NDSS)*, 2016.

[79] Executive Office of the President of the United States. The Cost of malicious cyber activity to the U.S. economy. https://www.whitehouse.gov/wp-content/uploads/2018/03/The-Cost-of-Malicious-Cyber-Activity-to-the-U.S.-Economy.pdf, 2018.

[80] N. Falliere, L. O. Murchu, and E. Chien. W32. Stuxnet dossier. *White paper, Symantec Corporation, Security Response*, 5(6), 2011.

[81] M. R. Farhadi, B. C. Fung, Y. B. Fung, P. Charland, S. Preda, and M. Debbabi. Scalable code clone search for malware analysis. *Digital Investigation*, 15:46–60, 2015.

[82] F. Farnstrom, J. Lewis, and C. Elkan. Scalability for clustering algorithms revisited. *ACM SIGKDD Explorations Newsletter*, 2(1):51–57, 2000.

[83] Q. Feng, M. Wang, M. Zhang, R. Zhou, A. Henderson, and H. Yin. Extracting conditional formulas for cross-platform bug search. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIACCS)*, pages 346–359. ACM, 2017.

[84] 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 (CCS)*, pages 480–491. ACM, 2016.

[85] A. Ferrari and A. Esuli. An NLP approach for cross-domain ambiguity detection in requirements engineering. *Automated Software Engineering (ASE)*, 26(3):559–598, 2019.

[86] H. Flake. Structural comparison of executable objects. In *Proceedings of the International GI SIG SIDAR Workshop on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 161–174. Gesellschaft für Informatik eV, 2004.

[87] A. Frank. On Kuhn's Hungarian method - A tribute from Hungary. *Naval Research Logistics (NRL)*, 52(1):2–5, 2005.

[88] E. Frank, Y. Wang, S. Inglis, G. Holmes, and I. H. Witten. Using model trees for classification. *Machine Learning*, 32(1):63–76, 1998.

[89] V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *International Conference on Computer Aided Verification (CAV)*, pages 519–531. Springer, 2007.

[90] D. Gao, M. K. Reiter, and D. Song. Binhunt: Automatically finding semantic differences in binary programs. In *International Conference on Information and Communications Security (ICICS)*, pages 238–255. Springer, 2008.

[91] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun. Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, pages 896–899. ACM, 2018.

[92] X. Gao, B. Xiao, D. Tao, and X. Li. A survey of graph edit distance. *Pattern Analysis and applications*, 13(1):113–129, 2010.

[93] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. Structural detection of Android malware using embedded call graphs. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security (AISec)*, pages 45–54. ACM, 2013.

[94] I. Goodfellow, Y. Bengio, and A. Courville. *Deep learning*. MIT press, 2016.

[95] G. Graefe and H. Kuno. Modern B-tree techniques. In *2011 IEEE 27th International Conference on Data Engineering (ICDE)*, pages 1370–1373. IEEE, 2011.

[96] C. Griffin. Graph Theory: Penn State Math 485 Lecture Notes. 2012.

[97] K. Griffin, S. Schneider, X. Hu, and T.-C. Chiueh. Automatic generation of string signatures for malware detection. In *International Workshop on Recent Advances in Intrusion Detection (RAID)*, pages 101–120. Springer, 2009.

[98] Y. Guillot and A. Gazet. Automatic binary deobfuscation. *Journal in computer virology*, 6(3):261–276, 2010.

[99] A. Gurfinkel. Testing: Coverage and structural coverage. University of Waterloo, https://ece.uwaterloo.ca/~agurfink/stqam/assets/pdf/W03-Coverage.pdf.

[100] G. Hamerly and C. Elkan. Learning the k in k-means. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 281–288, 2004.

[101] J. Han, J. Pei, and M. Kamber. *Data mining: concepts and techniques*. Elsevier, 2011.

[102] A. Heidarian and M. J. Dinneen. A hybrid geometric approach for measuring similarity level among documents and document clustering. In *IEEE Second International Conference on Big Data Computing Service and Applications (BigDataService)*, pages 142–151. IEEE, 2016.

[103] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra. Finding software license violations through binary code clone detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR)*, pages 63–72. ACM, 2011.

[104] S. Hido and H. Kashima. A linear-time graph kernel. In *Ninth IEEE International Conference on Data Mining (ICDM)*, pages 179–188. IEEE, 2009.

[105] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[106] S. Hochreiter and J. Schmidhuber. LSTM can solve hard long time lag problems. *Advances in Neural Information Processing Systems (NIPS)*, pages 473–479, 1997.

[107] G. Holmes, A. Donkin, and I. H. Witten. WEKA: A machine learning workbench. In *Proceedings of Australian New Zealnd Intelligent Information Systems Conference (ANZIIS)*, pages 357–361. IEEE, 1994.

[108] X. Hu, T.-c. Chiueh, and K. G. Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM conference on Computer and communications security (CCS)*, pages 611–620. ACM, 2009.

[109] Y. Hu, Y. Zhang, J. Li, and D. Gu. Binary code clone detection across architectures and compiling configurations. In *IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 88–98. IEEE, 2017.

[110] H. Huang, A. M. Youssef, and M. Debbabi. BinSequence: fast, accurate and scalable binary code reuse detection. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (ASIACCS)*, pages 155–166. ACM, 2017.

[111] D. Inc. Crashoverride: Analyzing the threat to electric grid operations. https://dragos.com/blog/crashoverride/CrashOverride-01.pdf, 2017.

[112] S. Ioffe. Improved consistent sampling, weighted minhash and l1 sketching. In *2010 IEEE International Conference on Data Mining (ICDM)*, pages 246–255. IEEE, 2010.

[113] E. R. Jacobson, A. R. Bernat, W. R. Williams, and B. P. Miller. Detecting code reuse attacks with a model of conformant program execution. In *Engineering Secure Software and Systems (ESSoS)*, pages 1–18. Springer, 2014.

[114] 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 (PASTE)*, pages 1–8. ACM, 2011.

[115] A. K. Jain. Data clustering: 50 years beyond k-means. *Pattern recognition letters*, 31(8):651–666, 2010.

[116] S. Jean, K. Cho, R. Memisevic, and Y. Bengio. On using very large target vocabulary for neural machine translation. *arXiv preprint arXiv:1412.2007*, 2014.

[117] R. Jhala and R. Majumdar. Path slicing. In *ACM SIGPLAN Notices*, volume 40, pages 38–47. ACM, 2005.

[118] J. Jiang, G. Li, M. Yu, G. Li, C. Liu, Z. Lv, B. Lv, and W. Huang. Similarity of binaries across optimization levels and obfuscation. In *European Symposium on Research in Computer Security (ESORICS)*, pages 295–315. Springer, 2020.

[119] W. Jin, S. Chaki, C. Cohen, A. Gurfinkel, J. Havrilla, C. Hines, and P. Narasimhan. Binary function clustering using semantic hashes. In *2012 11th International Conference on Machine Learning and Applications (ICMLA)*, volume 1, pages 386–391. IEEE, 2012.

[120] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin. Obfuscator-LLVM: software protection for the masses. In *Proceedings of the 1st International Workshop on Software PROtection (SPRO)*, pages 3–9. IEEE Press, 2015.

[121] M. G. Kang, P. Poosankam, and H. Yin. Renovo: A hidden code extractor for packed executables. In *Proceedings of the 2007 ACM Workshop on Recurring Malcode (WORM)*, pages 46–53. ACM, 2007.

[122] M. E. Karim, A. Walenstein, A. Lakhotia, and L. Parida. Malware phylogeny generation using permutations of code. *Journal in Computer Virology*, 1(1-2):13–23, 2005.

[123] H. Kashima and A. Inokuchi. Kernels for graph classification. In *IEEE International Conference on Data Mining (ICDM)*, volume 2002, 2002.

[124] W. M. Khoo. Decompilation as search. Technical report, University of Cambridge, Computer Laboratory, 2013.

[125] 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)*, pages 329–338. IEEE Press, 2013.

[126] S. Kim, M. Faerevaag, M. Jung, S. Jung, D. Oh, J. Lee, and S. K. Cha. Testing intermediate representations for binary analysis. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 353–364. IEEE Press, 2017.

[127] A. Kiss, J. Jász, G. Lehotai, and T. Gyimóthy. Interprocedural static slicing of binary executables. In *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM)*, pages 118–127. IEEE, 2003.

[128] E. B. Krissinel and K. Henrick. Common subgraph isomorphism detection by backtracking search. *Software: Practice and Experience*, 34(6):591–607, 2004.

[129] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *International Workshop on Recent Advances in Intrusion Detection (RAID)*, pages 207–226. Springer, 2005.

[130] C. Kruegel, W. Robertson, F. Valeur, and G. Vigna. Static disassembly of obfuscated binaries. In *USENIX security Symposium (USENIX Security 04)*, volume 13, pages 18–18, 2004.

[131] Y. Kwon, H. K. Kim, K. M. Koumadi, Y. H. Lim, and J. I. Lim. Automated vulnerability analysis technique for smart grid infrastructure. In *2017 IEEE Power & Energy Society Innovative Smart Grid Technologies Conference (ISGT)*, pages 1–5. IEEE, 2017.

[132] A. Lakhotia, M. D. Preda, and R. Giacobazzi. Fast location of similar code fragments using semantic 'juice'. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW)*, pages 1–6. ACM, 2013.

[133] A. M. Lamb, A. G. ALIAS PARTH GOYAL, Y. Zhang, S. Zhang, A. C. Courville, and Y. Bengio. Professor forcing: A new algorithm for training recurrent networks. *Advances in Neural Information Processing Systems (NeurIPS)*, 29:4601–4609, 2016.

[134] R. Langner. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security & Privacy (S&P)*, 9(3):49–51, 2011.

[135] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *In Proceedings of the international symposium on Code generation and optimization (CGO): feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[136] Q. Le and T. Mikolov. Distributed representations of sentences and documents. In *International conference on machine learning (ICML)*, pages 1188–1196, 2014.

[137] S. learn Developers. Mutual information in Python SKlearn, 2018.

[138] K. R. M. Leino. This is boogie 2. *Manuscript KRML*, 178(131), 2008.

[139] J. Leskovec, A. Rajaraman, and J. D. Ullman. *Mining of massive datasets*. Cambridge university press, 2014.

[140] D. Lin and M. Stamp. Hunting for undetectable metamorphic viruses. *Journal in computer virology*, 7(3):201–214, 2011.

[141] Z. Lin, M. Feng, C. N. d. Santos, M. Yu, B. Xiang, B. Zhou, and Y. Bengio. A structured self-attentive sentence embedding. *arXiv preprint arXiv:1703.03130*, 2017.

[142] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, and W. Zou. αdiff: Cross-version binary code similarity detection with dnn. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, pages 667–678. ACM, 2018.

[143] M. Liu, Y. Zhang, J. Li, J. Shu, and D. Gu. Security analysis of vendor customized code in firmware of embedded device. In *12th EAI International Conference on Security and Privacy in Communication Networks (SecureComm)*, 2016.

[144] L. Livi and A. Rizzi. The graph matching problem. *Pattern Analysis and Applications*, 16(3):253–283, 2013.

[145] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 389–400. ACM, 2014.

[146] L. Luo, J. Ming, D. Wu, P. Liu, and S. Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection. *IEEE Transactions on Software Engineering (TSE)*, 43(12):1157–1177, 2017.

[147] M.-T. Luong, H. Pham, and C. D. Manning. Effective approaches to attention-based neural machine translation. *arXiv preprint arXiv:1508.04025*, 2015.

[148] R. Mackiewicz. Overview of iec 61850 and benefits. In *2006 IEEE Power Engineering Society General Meeting (PES GM)*, pages 8–pp. IEEE, 2006.

[149] M. Marschalek. Big game hunting: Nation-state malware research. https://www.blackhat.com/docs/webcast/08202015-big-game-hunting.pdf/, 2015.

[150] L. Martignoni, M. Christodorescu, and S. Jha. OmniUnpack: Fast, generic, and safe unpacking of malware. In *Twenty-Third Annual Computer Security Applications Conference (ACSAC)*, pages 431–441. IEEE, 2007.

[151] L. Massarelli, G. A. Di Luna, F. Petroni, R. Baldoni, and L. Querzoni. SAFE: Self-attentive function embeddings for binary similarity. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 309–329. Springer, 2019.

[152] T. J. McCabe. A complexity measure. *IEEE Transactions on software Engineering (TSE)*, (4):308–320, 1976.

[153] A. McCallum. Efficiently inducing features of conditional random fields. In *Proceedings of the Nineteenth conference on Uncertainty in Artificial Intelligence (UAI)*, pages 403–410. Morgan Kaufmann Publishers Inc., 2002.

[154] J. J. McGregor. Backtrack search algorithms and the maximal common subgraph problem. *Software: Practice and Experience*, 12(1):23–34, 1982.

[155] B. D. McKay. Practical graph isomorphism. 1981.

[156] B. Medlock. An Introduction to NLP-based Textual Anonymisation. In *International Conference on Language Resources and Evaluation (LREC)*, pages 1051–1056. Citeseer, 2006.

[157] X. Meng. Fine-grained binary code authorship identification. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 1097–1099. ACM, 2016.

[158] X. Meng, B. P. Miller, and K.-S. Jun. Identifying multiple authors in a binary program. In *European Symposium on Research in Computer Security (ESORICS)*, pages 286–304. Springer, 2017.

[159] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 3111–3119, 2013.

[160] N. Moran and J. Bennett. *Supply Chain Analysis: From Quartermaster to Sun-shop*, volume 11. FireEye Labs, 2013.

[161] S. Muchnick. *Advanced compiler design implementation*. Morgan Kaufmann, 1997.

[162] J. Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics (SIAM)*, 5(1):32–38, 1957.

[163] G. Myles and C. Collberg. K-gram based software birthmarks. In *Proceedings of the 2005 ACM symposium on Applied Computing (SAC)*, pages 314–318. ACM, 2005.

[164] J. Nazario. BlackEnergy DDoS bot analysis. *Arbor Networks*, 2007.

[165] G. Neichin, D. Cheng, S. Haji, J. Gould, D. Mukerji, and D. Hague. 2010 US smart grid vendor ecosystem. 2010.

[166] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[167] M. Newman. *Networks: an introduction*. Oxford university press, 2010.

[168] A. Y. Ng, M. I. Jordan, and Y. Weiss. On spectral clustering: Analysis and an algorithm. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 849–856, 2002.

[169] B. H. Ng and A. Prakash. Expose: Discovering potential binary code re-use. In *37th Annual Computer Software and Applications Conference (COMPSAC)*, pages 492–501. IEEE, 2013.

[170] NIST. NIST/SEMATECH e-handbook of statistical methods. https://www.itl.nist.gov/div898/handbook/, 2013.

[171] J. Oliver, C. Cheng, and Y. Chen. TLSH - a locality sensitive hash. In *Fourth Cybercrime and Trustworthy Computing Workshop (CTC)*, pages 7–13. IEEE, 2013.

[172] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *the Journal of Machine Learning Research (JMLR)*, 12:2825–2830, 2011.

[173] H. Peng, F. Long, and C. Ding. Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy. *IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)*, 27(8):1226–1238, 2005.

[174] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. Cross-architecture bug search in binary executables. In *IEEE Symposium on Security and Privacy (S&P)*, pages 709–724. IEEE, 2015.

[175] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow. Leveraging semantic signatures for bug search in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference (CASAC)*, pages 406–415. ACM, 2014.

[176] O. Pourret, P. Naïm, and B. Marcot. *Bayesian networks: a practical guide to applications*. John Wiley & Sons, 2008.

[177] J. Qiu, X. Su, and P. Ma. Using reduced execution flow graph to identify library functions in binary code. *IEEE Transactions on Software Engineering (TSE)*, 42(2):187–202, 2016.

[178] B. B. Rad, M. Masrom, and S. Ibrahim. Opcodes histogram for classifying metamorphic portable executables malware. In *International Conference on e-Learning and e-Technologies in Education (ICEEE)*, pages 209–213. IEEE, 2012.

[179] A. Rahimian, P. Charland, S. Preda, and M. Debbabi. RESource: a framework for online matching of assembly with open source code. In *International Symposium on Foundations and Practice of Security (FPS)*, pages 211–226. Springer, 2012.

[180] A. Rahimian, P. Shirani, S. Alrabaee, L. Wang, and M. Debbabi. BinComp: A stratified approach to compiler provenance attribution. *Digital Investigation*, 14:S146–S155, 2015.

[181] M. Ramaswami and R. Bhaskaran. A study on feature selection techniques in educational data mining. *arXiv preprint arXiv:0912.3924*, 2009.

[182] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov. Learning and classification of malware behavior. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 108–125. Springer, 2008.

[183] D. Roobaert, G. Karakoulas, and N. V. Chawla. Information Gain, Correlation and Support Vector Machines. In *Feature Extraction*, pages 463–470. Springer, 2006.

[184] N. Rosenblum, B. P. Miller, and X. Zhu. Recovering the toolchain provenance of binary code. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, pages 100–110. ACM, 2011.

[185] N. Rosenblum, X. Zhu, and B. P. Miller. Who wrote this code? identifying the authors of program binaries. In *European Symposium on Research in Computer Security (ESORICS)*, pages 172–189. Springer, 2011.

[186] N. E. Rosenblum. *The Provenance Hierarchy of Computer Programs*. PhD thesis, University of Wisconsin–Madison, 2011.

[187] N. E. Rosenblum, B. P. Miller, and X. Zhu. Extracting compiler provenance from program binaries. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 21–28. ACM, 2010.

[188] I. Series. Business blackout, 2015.

[189] P. Shirani, L. Collard, B. L. Agba, B. Lebel, M. Debbabi, L. Wang, and A. Hanna. BinARM: Scalable and efficient detection of vulnerabilities in firmware images of intelligent electronic devices. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 114–138. Springer, 2018.

[190] P. Shirani, L. Wang, and M. Debbabi. BinShape: Scalable and robust binary library function identification using function shape. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 301–324. Springer, 2017.

[191] Y. Shoshitaishvili. *Building a base for cyber-autonomy*. PhD thesis, University of California, Santa Barbara, 2017.

[192] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna. Firmalice - automatic detection of authentication bypass vulnerabilities in binary firmware. In *the Network and Distributed System Security Symposium (NDSS)*, 2015.

[193] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy (S&P)*, pages 138–157. IEEE, 2016.

[194] X. Shu, D. Yao, and N. Ramakrishnan. Unearthing stealthy program attacks buried in extremely long execution paths. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 401–413. ACM, 2015.

[195] O. Shwartz, Y. Mathov, M. Bohadana, Y. Elovici, and Y. Oren. Opening Pandora's box: Effective techniques for reverse engineering IoT devices. In *International Conference on Smart Card Research and Advanced Applications, CARDIS*, pages 1–21. Springer, 2017.

[196] J. Slowik. Anatomy of an attack: Detecting and defeating CRASHOVERRIDE. *Virus Bulletin (VB2018), October*, 2018.

[197] A. Slowinska, T. Stancescu, and H. Bos. Body armor for binaries: Preventing buffer overflows without recompilation. In *USENIX Annual Technical Conference (ATC)*, pages 125–137, 2012.

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

[199] G. Taha. Counterattacking the packers. *McAfee Avert Labs, Aylesbury, UK*, 2007.

[200] J. Tekli, R. Chbeir, and K. Yetongnon. Efficient XML Structural Similarity Detection using Sub-tree Commonalities. In *Brazilian Symposium on Databases (SBBD)*, pages 116–130, 2007.

[201] K. Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, 1984.

[202] A. H. Toderici and M. Stamp. Chi-squared distance and metamorphic virus detection. *Journal of Computer Virology and Hacking Techniques*, 9(1):1–14, 2013.

[203] R. Torrance and D. James. The state-of-the-art in IC reverse engineering. In *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, pages 363–381. Springer, 2009.

[204] J. R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, 1976.

[205] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, pages 5998–6008, 2017.

[206] S. V. N. Vishwanathan, N. N. Schraudolph, R. Kondor, and K. M. Borgwardt. Graph kernels. *The Journal of Machine Learning Research (JMLR)*, 11:1201–1242, 2010.

[207] R. A. Wagner and M. J. Fischer. The string-to-string correction problem. *Journal of the ACM (JACM)*, 21(1):168–173, 1974.

[208] T. Wang, T. Wei, Z. Lin, and W. Zou. IntScope: Automatically detecting integer overflow vulnerability in x86 binary using symbolic execution. In *the Network and Distributed System Security Symposium (NDSS)*, 2009.

[209] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering (ICSE)*, pages 439–449. IEEE Press, 1981.

[210] H. Wen, Z. Lin, and Y. Zhang. FirmXRay: Detecting bluetooth link layer vulnerabilities from bare-metal firmware. In *Proceedings of the 27th ACM SIGSAC conference on Computer and communications security (CCS)*, pages 167–180. ACM, 2020.

[211] D. Wermke, N. Huaman, Y. Acar, B. Reaves, P. Traynor, and S. Fahl. A large scale investigation of obfuscation use in google play. *arXiv preprint arXiv:1801.02742*, 2018.

[212] S. Wold, K. Esbensen, and P. Geladi. Principal component analysis. *Chemometrics and intelligent laboratory systems*, 2(1-3):37–52, 1987.

[213] 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 (CCS)*, pages 363–376. ACM, 2017.

[214] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, and F. Song. SPAIN: Security patch analysis for binaries towards understanding the pain and pills. In *Proceedings of the 39th International Conference on Software Engineering (ICSE)*, pages 462–472. IEEE Press, 2017.

[215] C. Yang, Z. Liu, D. Zhao, M. Sun, and E. Y. Chang. Network representation learning with rich text information. In *International Joint Conference on Artificial Intelligence (IJCAI)*, volume 2015, pages 2111–2117, 2015.

[216] D. Yogatama, M. Faruqui, C. Dyer, and N. Smith. Learning word representations with hierarchical sparse coding. In *International Conference on Machine Learning (ICML)*, pages 87–96, 2015.

[217] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti. AVATAR: A Framework to Support Dynamic Security Analysis of Embedded Systems' Firmwares. In *The Network and Distributed System Security Symposium (NDSS)*, 2014.

[218] J. Zaddach and A. Costin. Embedded devices security and firmware reverse engineering. *Black-Hat USA*, 2013.

[219] F. Y. M. K. A. L. Z. Z. X. . X. D. Zeng, J. Obfuscation resilient binary code reuse through trace-oriented programming. In *ACM SIGSAC conference on Computer & communications security (CCS)*, pages 487–498. ACM, 2013.

[220] J. Zeng, Y. Fu, K. A. Miller, Z. Lin, X. Zhang, and D. Xu. Obfuscation resilient binary code reuse through trace-oriented programming. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS)*, pages 487–498. ACM, 2013.

[221] M. Zhang and R. Sekar. Control flow integrity for COTS binaries. In *USENIX Security Symposium (USENIX Security 13)*, pages 337–352, 2013.

[222] Y. Zheng, W. Ott, C. Gupta, and D. Graur. A scale-free method for testing the proportionality of branch lengths between two phylogenetic trees. *arXiv preprint arXiv:1503.04120*, 2015.

[223] R. Zhu, B. Zhang, J. Mao, Q. Zhang, and Y.-a. Tan. A methodology for determining the image base of arm-based industrial control system firmware. *International Journal of Critical Infrastructure Protection (IJCIP)*, 16:26–35, 2017.

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