Scalable analysis for malware and vulnerability detection in binaries

Chandramohan, Mahinthan

2018

Chandramohan, M. (2018). Scalable analysis for malware and vulnerability detection in binaries. Doctoral thesis, Nanyang Technological University, Singapore.

<https://hdl.handle.net/10356/80724>

<https://doi.org/10.32657/10220/46626>

NANYANG TECHNOLOGICAL UNIVERSITY

**Scalable Analysis for Malware and
Vulnerability Detection in Binaries**

Mahinthan Chandramohan

School of Computer Science and Engineering

A thesis submitted to Nanyang Technological University
in partial fulfilment of the requirements for the degree of
Doctor of Philosophy

September, 2018

THESIS ABSTRACT

Scalable Analysis for Malware and Vulnerability Detection in Binaries

by

Mahinthan Chandramohan

Doctor of Philosophy

School of Computer Science and Engineering

Nanyang Technological University, Singapore

In recent years, malware (malicious software) has greatly evolved and has become very sophisticated. The evolution and the volume of malware make it difficult to analyze them in an effective and scalable manner. It is reported that, on average, more than 100,000 fresh malware binaries emerge each day. Despite the common use and widespread availability of various anti-malware tools, proliferation of malware poses a major security threat to computer systems. Identification of malware variants significantly improves the detection rate and reduces the size of malware signature databases. Thus, it has become crucial for security researchers to analyze and cluster malware samples based on their malicious behavior. Though manual analysis of malware may provide a better understanding of its malicious behavior, it is practically impossible due to the exponential growth in the number of new malware each day. As a result, security researchers are constantly playing the catch-up game with the malware authors.

On the other hand, it is understood that software vulnerabilities are paving the way for malicious programs to find their targets, where attackers exploit the newly discovered vulnerabilities in the software systems to deliver their malicious payload. It is reported that more than 10,000 new vulnerabilities were reported in the first half of 2017. However, it is almost impossible to avoid vulnerabilities at the development stage (i.e., source code level); it is even difficult to discover vulnerabilities at the production stage (i.e., machine code level). Further, there are still lots of software applications in the wild with known security vulnerabilities that are unpatched even when the official patches are available. In most cases, attackers leverage on exploits written for such unpatched vulnerabilities to compromise the targets and achieve their malicious intent. Identifying the unpatched vulnerabilities and alerting the interested parties in a timely manner helps to significantly reduce the malware proliferation rate. In addition to detecting unpatched vulnerabilities, it is also important to identify potential vulnerable locations (or *attack*

surfaces) within the software application that can turn into a zero-day (or 0-day) vulnerability in the future, which can be later exploited or weaponized by the malware authors to deliver malicious payloads.

The first part of this thesis is dedicated to addressing the scalability issues in the existing malware behaviour modelling techniques. In this work, we propose and evaluate a bounded feature space behaviour modelling technique, named eBOFM, for effective and scalable malware detection and triaging tasks. The malware features extracted using eBOFM are very expressive in nature and provide valuable information to security analysts. Most importantly, the eBOFM feature space is bounded by an upper limit. Whereas, in existing malware behaviour modelling techniques, either the feature space grows in proportion to the size of the execution traces (e.g., n-gram models) or high computational power and memory is required (e.g., graph-based models), which makes them impractical to use in real-world malware detectors and triaging tools. We also perform large-scale experiments showing that, with eBOFM, computation time and memory usage are vastly lower than currently reported results in existing malware detection and clustering techniques, while preserving or even improving their high detection and clustering accuracy.

The second part of this thesis is dedicated to addressing the challenges in identifying the security patches at machine code (or binary) level. Understanding of vulnerabilities and their corresponding patches at binary level allows us to generate better vulnerability signatures that are later used to detect unpatched vulnerabilities in the commercial off-the-shelf (COTS) binaries. In this work, we propose a scalable binary-level patch analysis framework, named SPAIN, which can automatically identify security patches and summarize patch patterns and their corresponding vulnerability patterns. The summarized patterns can be used to search similar patches or vulnerabilities in binary programs. Our experimental results on several real-world projects have shown that SPAIN can identify security patches with high accuracy and scalability. Using SPAIN, we have summarized the vulnerability and the corresponding patch patterns for five major class of vulnerabilities including use-after-free, double free, integer under(over) flow, null pointer dereference and buffer overflow.

The third part of this thesis is dedicated to addressing the challenges in performing cross-architecture cross-platform binary matching. Architecture and platform agnostic binary matching is a vital component in detecting unpatched vulnerabilities, where vulnerability patterns are obtained using patch analysis (in our case,

SPAIN) and they are matched against the binaries in the wild. Different from source code clone detection, clone detection (similar code search) in binary executables faces big challenges due to the significant differences in the syntax and the structure of binary code that result from different configurations of compilers, architectures and OSs. To address these challenges, in this work, we propose BINGO – a scalable and robust binary search engine supporting various architectures and operating systems. The key contributions include binary emulation and selective function inlining technique to capture the complete program semantics. Further, architecture and OS neutral function filtering is proposed to improve the scalability, where it removes the irrelevant functions from the matching process. Besides, we also introduce variant length partial traces to model binary functions in a program structure agnostic fashion. The experimental results show that BINGO can find semantic similar functions across architecture and OS boundaries, even with the presence of program structure distortion, in a scalable manner. Using BINGO, we also discovered a zero-day vulnerability in Adobe PDF Reader, a COTS binary.

The final part of this thesis is dedicated to identifying potential vulnerable functions for (open source) software, where identifying potentially vulnerable locations in a code base is critical a pre-step for effective vulnerability assessment in the binary level; In this work, we propose a light weight approach to identify vulnerable functions through program analysis, which requires no prior knowledge about the application. In this way, we are completing our approach by a complete solution to find vulnerabilities of the application via both matching based method for known vulnerabilities and program analysis based method for unknown vulnerabilities.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to my supervisors Prof. Yang Liu and Prof. Thambipillai Srikanthan for their guidance and continuous support throughout my doctoral journey. They have been very encouraging, supportive and very approachable, and always there for me when I am in need. They also helped me to pass through the most difficult time in my research. Their guidance really helped me to shape my research, where they encouraged critical thinking and gave me ample of opportunities to interact and collaborate with various industry and academic partners.

I am also very grateful to Dr. Bihuan Chen, Dr. Annamalai Narayan, Dr. Guozhu Meng, and Dr. Yinxing Xue for their valuable advice and suggestions that helped me to improve my research work. Further, they also helped to improve my writing skills.

In addition, I would like to take this opportunity to thank my friends and colleagues at the Cyber Security Lab (CSL) for their support and making my doctoral journey a pleasant experience. Further, I would also like to thank CSL laboratory executive Mr. Tan Suan Hai for his help and technical support during my stay at Cyber Security Lab.

Finally, I am indebted to my parents for their unconditional support and love, and always encouraging me for my studies and raising me to the level where I stand today.

Contents

1	Introduction	1
1.1	Motivation and Goals	2
1.2	Contributions	3
1.3	Publication List	7
2	Scalable Malware Detection and Triaging Through Bounded Feature Space Behavior Modeling	9
2.1	Introduction	9
2.2	Methodology	14
2.2.1	Operating System (OS) Resources	14
2.2.2	Malware Feature Representation	17
2.2.3	Security-critical Attributes	18
2.2.4	Properties of eBOFM features	21
2.2.5	Example	22
2.2.6	Theoretical Upper-bound Calculation	23
2.3	Construction of Feature Vectors	25
2.3.1	Collecting Execution Traces	25
2.3.2	Extraction of Features	25
2.3.3	Embedding Malware Features	26
2.4	Detection Method	26
2.5	Clustering Method	27
2.6	Experimental Design	28
2.6.1	Experimental dataset	28
2.6.2	Evaluation Criteria	31
2.6.3	Experiment Termination Criteria	32
2.6.4	Evaluation Measures	34
2.6.5	Experimental Setup	34
2.7	Experimental Results	35
2.7.1	Malware Detection Experiment	35
2.7.2	Malware Clustering Experiment	37
2.7.3	Scalability of eBOFM Models	39
2.7.4	Qualitative Analysis of eBOFM Features	41
2.8	Related Work	44
2.8.1	Behaviour Modelling for Malware Clustering	44
2.8.2	Behavior Modelling for Malware Detection	46

2.9	Conclusion	48
3	Security Patch Analysis for Binaries - Towards Understanding the Pain and Pills	51
3.1	Introduction	51
3.2	Preliminaries and Overview	54
3.2.1	Preliminaries	54
3.2.2	Running Example	55
3.2.3	Framework Overview	56
3.2.4	Assumptions	58
3.3	Methodology	59
3.3.1	Locating Patched Functions	59
3.3.1.1	Function Matching	59
3.3.1.2	Function Filtering	60
3.3.2	Identifying Patched Basic Blocks	61
3.3.3	Identifying Security Patches	63
3.3.4	Summarizing Patch and Vulnerability Patterns	66
3.3.5	Applications of Patterns	67
3.4	Evaluation	68
3.4.1	Accuracy and Scalability (RQ1)	69
3.4.1.1	Accuracy	69
3.4.1.2	Scalability	72
3.4.2	Patch and Vulnerability Patterns (RQ2)	73
3.4.3	Application of Patterns (RQ3)	76
3.5	Discussion	77
3.6	Related Work	78
3.6.1	Patch Analysis and Diffing	78
3.6.2	Vulnerability Modeling and Searching	79
3.7	Conclusion	81
4	Accurate and Scalable Cross-Architecture Cross-OS Binary Code Search with Emulation	83
4.1	Introduction	83
4.2	Background and Motivation	90
4.2.1	Background	90
4.2.2	Motivating Example	93
4.2.3	Challenges for Syntactic Approaches	93
4.2.4	Challenges for Semantics-based Matching	94
4.3	System Overview of BINGO-E	95
4.3.1	The Sketch of Work Flow	96
4.3.2	Solutions to the Matching Challenges	98
4.4	Binary Code Matching Features	99
4.4.1	Structural Features	100
4.4.2	High-level Semantic Features	102
4.4.3	Low-level Semantic Features	104

4.4.3.1	Features Used in BINGO	104
4.4.3.2	Features Newly Added in BINGO-E	106
4.4.3.3	Emulation	107
4.5	Accurate: Selective Inlining	112
4.5.1	Function Invocation Patterns	112
4.5.2	Inline Decision Algorithm	113
4.6	Scalable Function Matching	115
4.6.1	Primary Matching	115
4.6.2	Low-level Semantic Features Matching	116
4.6.2.1	K-length Partial Trace Extraction	117
4.6.2.2	Function Model Generation	117
4.6.3	Final Matching	119
4.7	Experimental Results	120
4.7.1	Research Question	121
4.7.2	Answer to RQ1: Robustness in Cross-architecture Matching	122
4.7.2.1	BINGO's Answer to RQ1	122
4.7.2.2	BINGO-E's Answer to RQ1	124
4.7.3	Answer to RQ2: Robustness in Cross-compiler Matching	127
4.7.3.1	BINGO's Answer to RQ2	127
4.7.3.2	BINGO-E's Answer to RQ2	129
4.7.4	Answer to RQ3: Robustness in Cross-platform (OS) Matching	130
4.7.4.1	BINGO's Answer to RQ3	130
4.7.4.2	BINGO-E's Answer to RQ3	132
4.7.5	Answer to RQ4: Applications	134
4.7.5.1	BINGO's Answer to RQ4: Vulnerability Detection via Matching Open-source Software Bug to Com- mercial Software	134
4.7.5.2	BINGO-E's Answer to RQ4: New Application of Matching Binaries of Forked Projects	135
4.7.6	Answer to RQ5: Solving the Typical Code Clone Search Problem with the Same OS, Architecture, Compiler	138
4.7.7	Answer to RQ6: Scalability	140
4.7.7.1	BINGO's Answer to RQ6	140
4.7.7.2	BINGO-E's Answer to RQ6: Better Scalability	141
4.7.8	Discussion	141
4.7.8.1	Threats to Validity	141
4.7.8.2	The Focus of Our Study.	143
4.8	Related Work	144
4.8.1	Code Search Using Low-level Program Semantics	144
4.8.2	Code Search Using High-level Program Semantics	145
4.8.3	Code Search Using Structural Information	146
4.8.4	Security Application of Binary Code Search	147
4.9	Conclusion	148

5	Identifying Potential Vulnerable Functions via Program Analysis	151
5.1	Introduction	151
5.2	Methodology	153
5.2.1	Overview	153
5.2.2	Function Binning	155
5.2.3	Function Ranking	157
5.3	Evaluation	159
5.3.1	Evaluation Setup	159
5.3.2	Rationality of Binning-and-Ranking	161
5.3.3	Effectiveness of Binning-and-Ranking	162
5.3.4	Sensitivity of the Metrics	166
5.3.5	Predictability of the Metrics	168
5.3.6	Scalability of Our Framework	170
5.3.7	Applications of Our Framework	170
5.4	Conclusions	172
6	Conclusions and Future Research	173
	Bibliography	177

List of Figures

2.1	BOFM and eBOFM features extracted for the sample malware trace shown in listing:5	23
2.2	Sample malware feature vector	26
2.3	(a) shows the Intra-family analysis of Allapple Worm. (b) shows the Inter-family analysis between Looper Trojan, Allapple worm and AdRotator adware. Darker area represents active malware	42
3.1	Running Example: NULL Pointer Dereference Vulnerability and its Patch from OpenSSL 1.0.1l.	55
3.2	Framework Overview of SPAIN	56
3.3	An Example of Compiler-Introduced Changes	60
3.4	An Example of the Original and Patched Functions	64
3.5	An Example of Pre-State and Post-State	66
3.6	Abstract Vulnerability and Security Patch Pattern	67
4.1	Code segment responsible for Heartbleed vulnerability (CVE-2014-0160) appeared as in the binary (a) compiled with gcc 4.6, and (b) compiled with mingw32	92
4.2	A false positive case of BINGO	95
4.3	BINGO-E work flow: rectangles represent processing steps, ovals represent features, arrows represent control/data flows	96
4.4	The BB-structure of a function, and its structural features	102
4.5	the high-level semantic features of an exemplar function	109
4.6	An example of using emulation to extract low-level features	110
4.7	Commonly observed function invocation patterns, where ‘UD’ denotes user-defined callee function. Here, all the incoming and outgoing edges (or calls) represent user-defined functions, unless specified as <i>Libcall</i> or <i>Termination Libcall</i> next to them	111
4.8	A sample CFG (a) and the extracted 3-length (b) and 2-length (c) partial traces	117
4.9	A sample signature (a) and target (b) functions, where the lines indicate the matched partial traces and the nodes refer to the basic-blocks.	118
4.10	In BINGO, cross-architecture function matching before/after selective inlining using <i>coreutils</i> binaries. Here, C and G represent compilers <i>clang</i> and <i>gcc</i> , respectively, and 32 and 64 denote x86 32bit and 64bit architectures.	123

4.11	Cross-architecture function matching after selective inlining using <code>coreutils</code> binaries for BINGO and BINGO-E. Here, C and G represent compilers <code>clang</code> and <code>gcc</code> , respectively, and 32 and 64 denote x86 32bit and 64bit architectures.	125
5.1	An Overview of the Proposed Framework	154
5.2	Vulnerability Score vs. Complexity Score for Non-Vulnerable Functions (in Blue) and Vulnerable Functions (in Red)	161
5.3	Percentage of Functions (<i>Iden. Func.</i>) that are Identified as Potentially Vulnerable, and Percentage of Vulnerable Functions (<i>Cov. Vul. Func.</i>) and CVEs (<i>Cov. CVE</i>) that are Covered by Those Identified Potentially Vulnerable Functions as k Increases	163
5.4	Sensitivity Analysis Results of the Complexity Metrics	167
5.5	Predictability Analysis Results of the Three Projects	170

List of Tables

2.1	Possible actions that can be performed on Registry resource	16
2.2	Comparison of execution traces for simulated user and real-world user interactions	30
2.3	Summary of ground truth clusters. Number of clusters generated based on the malware labels assigned by each AV vendor	34
2.4	Summary of Malware Detection Accuracy achieved by SVM.	36
2.5	Clustering accuracy achieved by PBCA for mal-small	38
2.6	Clustering accuracy achieved by MBKM for mal-small	38
2.7	Clustering accuracy achieved by PBCA for mal-large	38
2.8	Summary of Malware features generated	41
2.9	Memory requirements (in Megabytes) for each malware behaviour modelling technique	41
2.10	Time requirements (in seconds) for each malware behaviour modelling technique	42
3.1	Accuracy and Performance on OpenSSL	70
3.2	Accuracy on OpenSSL w.r.t Vulnerability Types	71
3.3	Performance on Linux and Adobe PDF Reader	73
3.4	Patch Patterns vs. Vulnerability Patterns, where TNT denotes taint, IN denotes input, and SSTV denotes sensitive.	75
4.1	Comparison of existing techniques. Here, ✓ or ✗ represent whether it supports the corresponding feature or not, respectively.	86
4.2	Different categories of features used by BINGO-E	98
4.3	Total no. of functions selectively inlined in coreutils binaries compiled using gcc and clang for ARM, x86 32bit and 64bit architectures.	124
4.4	In BINGO, percentage of functions ranked #1 in inter-compiler (x86 32bit) comparison for coreutils . Here, C and G represent clang and gcc compilers, respectively and 0-3 represents optimization O0-O3.	128
4.5	In BINGO, percentage of functions ranked #1 in intra-compiler (x86 32bit) comparison for coreutils . Here, G and G' represent gcc v4.8.3 and v4.6 compilers, respectively.	129

4.6	In BINGO-E, percentage of functions ranked #1 in inter-compiler (x86 32bit) comparison for <code>coreutils</code> . Here, C and G represent <code>clang</code> and <code>gcc</code> compilers, respectively and 0-3 represents optimization O0-O3.	130
4.7	In BINGO-E, percentage of functions ranked #1 in intra-compiler (x86 32bit) comparison for <code>coreutils</code> . Here, G and G' represent <code>gcc</code> v4.8.3 and v4.6 compilers, respectively.	131
4.8	BINGO's returned rankings (top-10 matches) for 60 matched cases in <code>msvcrt</code> against <code>libc</code>	133
4.9	BINGO-E's returned rankings (top-10 matches) for 60 matched cases in <code>msvcrt</code> against <code>libc</code>	133
4.10	In BINGO-E, percentage of functions ranked #1 in inter-compiler (x86 32bit) comparison of <code>thttpd</code> to <code>sthttpd</code> . Here, C and G represent <code>clang</code> and <code>gcc</code> compilers, respectively and 0-3 represents optimization O0-O3. Bold font highlights results of the configuration <code>gcc</code> O0-O2, the same as that in CoP [1].	137
4.11	In BINGO-E, percentage of functions ranked #1 in inter-compiler (x86 32bit) comparison of <code>sthttpd</code> to <code>thttpd</code> . Here, C and G represent <code>clang</code> and <code>gcc</code> compilers, respectively and 0-3 represents optimization O0-O3. Bold font highlights results of the configuration <code>gcc</code> O0-O2, the same as that in CoP [1].	137
4.12	Matching results between functions of <code>BusyBox</code> 1.20.0 and functions of <code>BusyBox</code> 1.21 using BINGO-E, TRACY and BINDIFF	138
5.1	The Complexity Metrics of a Function	156
5.2	The Vulnerability Metrics of a Function	158
5.3	Details of the Target Applications	160
5.4	The Appropriate k for Each Project	163
5.5	Covered CVEs vs. Identified Vulnerable Functions	165
5.6	Identified Vulnerable Functions vs. Covered CVEs	166
5.7	Details of the Older Version and Four Later Versions of the Three Projects	169
5.8	Performance of Our Framework	172

1

Introduction

It is estimated that more than 100,000 fresh malware binaries emerge each day. Despite the common use and widespread availability of various anti-malware tools, proliferation of malware poses a major security threat to computer systems. The exponential growth of malware made manual analysis impossible. As a result, security researchers are constantly playing the catch-up game with the malware authors. One of the major reasons for this epidemic proliferation of malware is that there are huge number of vulnerable software systems/applications out there in the wild that are paving the way for malicious programs to find their targets, where attackers exploit the newly discovered (also known as 0-day or zero-day) or unpatched (also known as 1-day) vulnerabilities in the software applications to deliver their malicious payload. In the first half of 2017 alone, more than 10,000

zero-day vulnerabilities were publicly disclosed, while there are many more yet to be discovered and/or publicly disclosed.

1.1 Motivation and Goals

In the past, security researchers have concluded that static malware analysis techniques have several serious limitations [2]. For example, the inability to handle anti-reverse engineering techniques, such as obfuscation, packing and encryption, employed by malware authors to evade static malware detection engines. In addition, the lack of visibility upfront where, in modern malware managed by sophisticated command and control (C&C) infrastructure, the malicious payload is delivered in parts at various stages of its execution and hence, to get the complete picture of the malicious behaviour, it needs to be executed. Finally, toolchain support for static malware analysis is very limited where the source code for malware is not readily available and the disassembly/decompilation of malware binaries are error prone and ambiguous, where this task is further hindered by aforementioned anti-reverse engineering techniques.

To alleviate the aforementioned problems in static malware analysis, in the recent years, researchers have proposed behaviour-based malware analysis, where the malware is executed to understand its malicious behaviour. There exists a significant literature on behaviour-based malware analysis techniques for malware detection [3–7] and triaging [8–14]. Unfortunately, one of the key issues in existing behaviour-based malware analysis techniques is that they are not scalable. That is, the malware features (also known as *malware models*) extracted from the malware execution traces are unacceptably very huge and on the other hand, the feature extraction process (also known as *malware behaviour modelling technique*) itself is very computation and memory intensive such that does not scale well to meet the real-world demand.

On the other hand, it is understood that software vulnerabilities are paving the way for malicious programs to find their targets, where attackers exploit the newly discovered vulnerabilities in the software systems to deliver their malicious payload. However, it is almost impossible to avoid vulnerabilities at the development stage (i.e., source code level); it is even difficult to discover vulnerabilities at the production stage (i.e., machine code level). Unfortunately, there are still lots of software applications in the wild with known security vulnerabilities that are unpatched even when the official patches are available. In most cases, attackers leverage on exploits written for such unpatched (or 1-day) vulnerabilities to compromise the targets and achieve their malicious intent. Identifying the unpatched vulnerabilities and alerting the interested parties in a timely manner helps to significantly reduce the malware proliferation rate. In addition to detecting unpatched vulnerabilities, it is also important to identify potential vulnerable locations (or *attack surfaces*) within the software application that can turn into a zero-day (or 0-day) vulnerability in the future, which can be later exploited or weaponized by the malware authors to deliver malicious payloads.

1.2 Contributions

In the first part of this thesis, we address the scalability issues in the existing malware behaviour modelling techniques. In this work, we propose and evaluate a bounded feature space behaviour modelling technique, named eBOFM, for effective and scalable malware detection and triaging tasks. Our technique models the interactions between software (which can be malware or benign) and security-critical operating system (OS) resources in a systematic manner. The malware features extracted using eBOFM are very expressive in nature and provide valuable information to security analysts. Most importantly, the eBOFM feature space is bounded by an upper limit. Whereas, in existing malware behaviour modelling techniques, either the feature space grows in proportion to the size of the execution traces (e.g., n -gram models) or high computational power and memory is required

(e.g., graph-based models), which makes them impractical to use in real-world malware detectors and triaging tools

In summary, to enable effective malware detection and triaging, our work makes the following contributions.

- We propose a unified malware behavior modelling technique (called eBOFM) that captures malicious interactions between malware and security-critical OS resources in a scalable manner and at an adequate level of abstraction. When combined with machine learning techniques, eBOFM models can be directly used for automated malware detection and clustering tasks without any transformations.
- We conducted and report three experiments involving two malware datasets¹ containing 4,853 and 151,342 malware samples each. Our approach, when combined with appropriate machine learning classifiers, can achieve a malware detection rate of 91.7% with no false positives, the latter being important in our context. With respect to clustering, it can achieve an average accuracy (F1-score) of 93.9% and 62.2% on the small and large malware datasets, respectively, the latter lower score being due to a lower recall.
- We demonstrate the effectiveness of eBOFM features in malware detection and clustering operations by showing that eBOFM models dramatically reduce the malware feature spaces and yet improve the detection and clustering accuracies compared to four other malware behavior modelling techniques proposed in the literature.
- In addition, we also show that eBOFM models achieve a significant reduction in time (68%) and memory (72%) requirements against a well-known open-source malware clustering framework, Malheur [Rieck et al. 2011], when clustering the large dataset.

In the second part of this thesis, we address the challenges in identifying the security patches at machine code (or binary) level. Understanding of vulnerabilities

and their corresponding patches at binary level allows us to generate better vulnerability signatures that are later used to detect unpatched vulnerabilities in the commercial off-the-shelf (COTS) binaries. In this work, we propose a scalable binary-level patch analysis framework, named SPAIN, which can automatically identify security patches and summarize patch patterns and their corresponding vulnerability patterns. Specifically, given the original and patched versions of a binary program, we locate the patched functions, and identify the changed traces (i.e., a sequence of basic blocks) that may contain security or non-security patches. Then we identify security patches through a semantic analysis on these traces, and summarize the patterns through a taint analysis on the patched functions. The summarized patterns can be used to search similar patches or vulnerabilities in binary programs.

In summary, our patch analysis work makes the following contributions.

- We propose a scalable binary-level patch analysis framework SPAIN, which can identify security patches and summarize patch patterns and their corresponding vulnerability patterns.
- We implement SPAIN in a prototype and conducted experiments to demonstrate the accuracy, scalability, and application of SPAIN.
- Using SPAIN, we have summarized the vulnerability and the corresponding patch patterns for five major classes of vulnerabilities including use-after-free, double free, integer under(over) flow, null pointer dereference and buffer overflow.
- We discover undocumented security patches and found one zero-day vulnerabilities in Adobe PDF Reader.

In the third part of this thesis, we address the challenges in performing cross-architecture cross-platform binary matching. Architecture and platform agnostic binary matching is a vital component in detecting unpatched vulnerabilities, where vulnerability patterns are obtained using patch analysis (in our case, SPAIN) and

they are matched against the binaries in the wild. Different from source code clone detection, clone detection (similar code search) in binary executables faces big challenges due to the gigantic differences in the syntax and the structure of binary code that result from different configurations of compilers, architectures and OSs. Existing studies have proposed distinct categories of features for detecting binary code clones, including CFG structures, n-gram in CFG, input/output values, etc. However, they suffer from several limitations including incomplete program semantics and scalability. To address these limitations, in this work, we propose BINGO – a scalable and robust binary search engine supporting various architectures and operating systems. The key contributions include binary emulation and selective function inlining technique to capture the complete program semantics. Further, architecture and OS neutral function filtering is proposed to improve the scalability, where it removes the irrelevant functions from the matching process. Besides, we also introduce variant length partial traces to model binary functions in a program structure agnostic fashion. The experimental results show that BINGO can find semantic similar functions across architecture and OS boundaries, even with the presence of program structure distortion, in a scalable manner. Using BINGO, we also discovered a zero-day vulnerability in Adobe PDF Reader, a COTS binary.

In summary, our binary matching work makes the following contributions:

- We propose a selective inlining algorithm to capture the complete semantic of the binary functions.
- We introduce an architecture and OS neutral function filtering process that helps narrow down the target function search space and leverage on variant length partial traces to model the function at various granularity levels that is agnostic to underlying program structures.
- Our work is the first attempt to incorporate emulation with cross-architecture cross-OS binary code search.

- We empirically demonstrate that SPAIN outperforms the state-of-the-art binary function matching tools, and also report the zero-day vulnerability discovered from Adobe PDF Reader.

In the final part of this thesis, we propose an approach to identify potential vulnerable functions in (open source) software. Particularly, we develop a lightweight solution to identify vulnerable functions through program analysis, which requires no prior knowledge about the application.

1.3 Publication List

The list of publications of the Ph.D candidate is listed below:

1. Mahinthan Chandramohan and Hee Beng Kuan Tan. “Detection of mobile malware in the wild.” *IEEE Computer*, 45, 9, 65-71, 2012 [18].
2. Mahinthan Chandramohan, Hee Beng Kuan Tan and Lwin Khin Shar. “Scalable Malware Clustering through Coarse-Grained Behavior Modeling” *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering (FSE), (NIER Track)*, 2012. ACM [17].
3. Mahinthan Chandramohan, Hee Beng Kuan Tan, Lionel C Briand, Lwin Khin Shar and Padmanabhuni Bindu Madhavi. “A scalable approach for malware detection through bounded feature space behavior modeling.” *Proceedings of the 28th International Conference on Automated Software Engineering (ASE)*, 312-322, 2013, IEEE/ACM [10].
4. Sanjeev Das, Yang Liu, Wei Zhang, and Mahinthan Chandramohan. “Semantics-Based Online Malware Detection: Towards Efficient Real-Time Protection Against Malware.” *IEEE Transactions on Information Forensics and Security (TIFS)*, 11, no. 2, 289-302, 2016 [16].

5. Mahinthan Chandramohan, Yinxing Xue, Xu Zhengzi, Yang Liu, Chia Yuan Cho and Hee Beng Kuan Tan. “Bingo: cross-architecture cross-os binary search.” Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE), 678-689, 2016, ACM [111].
6. Yinxing Xue, Zhengzi Xu, Mahinthan Chandramohan and Yang Liu. “Accurate and Scalable Cross-Architecture Cross-OS Binary Code Search with Emulation.” IEEE Transactions on Software Engineering (TSE), (Accepted). 2018 [14]
7. Zhengzi Xu, Bihuan Chen, Mahinthan Chandramohan, Yang Liu and Fu Song. “SPAIN: Security Patch Analysis for Binaries - Towards Understanding the Pain and Pills.” Proceedings of the 39th International Conference on Software Engineering (ICSE 2017), Buenos Aires, Argentina, May 20-28, 2017, IEEE [15].

2

Scalable Malware Detection and Triaging Through Bounded Feature Space Behavior Modeling

2.1 Introduction

According to McAfee, an anti-malware vendor, AV scanners detected more than 650 million PC malware (called hereafter malware) samples targeting the Windows Operating System in 2013¹. On average, around 200,000 malware samples appear

¹McAfee Threat Report 2016: <https://bit.ly/2QSdptB>

each day². Despite the common use and widespread availability of various anti-malware tools, proliferation of malware poses a major security threat to computer systems. Identification of malware variants significantly improves the detection rate and reduces the size of malware signature databases. Thus, it has become crucial for anti-malware vendors to analyze and cluster malware samples based on their malicious behavior. Though manual analysis of malware may provide a better understanding of its malicious behavior, it is practically impossible due to the exponential growth of new malware variants in the recent years³.

Due to large volume of malware samples it is harder to handle and analyze them in a systematic manner. Thus, in malware triage, clustering plays a significant role [19, 20]. In addition, malicious software seen in the wild tends to be a variant of already known malware [21]. It is also reported that the top seven malware families account for more than 50% of all variants found and the top 25 families account for over 75% [22]. Thus, malware clustering is used as a pre-filtering process to group malware variants with similar behavioral patterns. This significantly reduces the number of actual malware samples that need to be analyzed by an expert to generate effective detection mechanisms, which are based on the identification of malicious features, e.g., malicious system call patterns.

There exists a significant literature on behavior-based malware clustering [8, 9, 11–13, 17, 23], and detection mechanisms [3–7] that rely on various behavior modeling techniques such as n-gram (or n-tuple) models, non-transient state changes, system call trace analysis, taint analysis and system call dependency graphs to extract malicious features. Unfortunately, one of the key issues in existing behavior modeling techniques is that the scalability of these approaches is highly problematic. Here, scalability refers to the number of malware features (or signatures) extracted from an execution trace. For example, even a simple model such as bags of system calls, generates a number of malware features that grows proportionally to the size of execution traces [5]. In addition to the scalability of feature spaces, unacceptably high computational complexity and memory consumption are also expected

²Kaspersky Whitepaper 2014: <https://bit.ly/2zoui8L>

³McAfee Threat Report 2013: <https://bit.ly/2MS9ppH>

to impact the practicality and efficiency of malware analysis tools. As a result, applications of complex malware behavior modeling techniques, such as system call dependency and behavioral graphs, are very limited in practice. For example, in [7], it is reported that it took 12-48 hours to extract malware specifications from a network worm using a graph mining algorithm.

However, one might argue that huge feature spaces may not have a significant impact on malware clustering, as it is an offline process (done at the backend of an anti-malware vendor), which can rely on more computing power or hardware support. In contrast to malware clustering, minimal feature space is vital for effective and scalable malware detection tools as detection happens at the end host, which has much less computing power [5]. Further, adding more computing power or hardware support can only be a temporary solution, considering the growth rate of new malware variants, e.g., an increase of 69% reported between Q2 2014 and Q2 2015⁴. Thus, it is essential to have a scalable malware behavior modeling technique that is capable of extracting interesting features from malware samples. Such features must provide valuable insight into malicious behavior to security analysts during the offline malware triaging process, must have a confined feature space, and must have acceptable computational power and memory requirements such that the extracted features can be directly used in a malware detector.

Another important limitation in the malware behavior modeling techniques proposed in the literature is that they are either used for malware detection or triaging but not both, where features used for detection, in general, tend to be concise, simple and easy to extract while the features used for triaging tend to be complex and more expressive about the malicious behavior. As a result, there is a semantic gap between these two types of malware features. However, bridging this gap (i.e., using a unified model for both tasks) would save enormous time and effort for security analysts. Most importantly, the discriminative (or interesting) features observed during malware triaging (or clustering) can be directly used as signatures in malware detection tools. In addition, having a concise, simple, easy to extract and yet, expressive malware behavior models will eliminate the detection errors

⁴McAfee Threats Report 2015: <https://bit.ly/1IIV7Ou>

and imprecision due to feature (or dimensionality) reduction techniques, such as hashing [9, 19, 24], that are used to address the curse of dimensionality problem.

To this end, we propose and evaluate a simple yet efficient malware behavior modeling technique called eBOFM that systematically captures the interactions between malware and security-critical system resources in a scalable manner and at an adequate level of abstraction. eBOFM is an extension of our earlier behavior modeling technique called BOFM [10] that is proposed for scalable malware detection. In our previous work, the interactions between malware and operating system resources are modelled at a very high-level of abstraction such that the security-critical information that is readily available from the system resources is entirely missing from the extracted malware features. However, we find these additional details provide valuable insights into malicious behavior to security analysts, an essential property during malware triaging, which in turn, helps to generate effective malware signatures.

Hence, the eBOFM features are constructed by annotating BOFM features with relevant security-critical attributes, where attributes provide a deeper understanding of the malicious behavior. For example, given a malware interacting with a file, eBOFM features help to address these critical questions: Is the file executable? What are the operations performed on the file (e.g., delete)? Where is the file located (e.g., system directory)? Similarly, for all malware interactions involving registry, process and network (to name a few), eBOFM features provide valuable insights into the actual malicious behavior.

In eBOFM, scalability is achieved through confining the malware feature space such that it is upper-bounded by a predetermined value N (Section Theoretical Upper-bound Calculation). That is, eBOFM can extract malware features that do not grow in proportion with the number of samples examined and, in addition, the processing of eBOFM features requires acceptable computation power and memory consumption, as demonstrated by our experiments. Further, eBOFM also minimizes uninteresting features (i.e., noise) and is resilient against basic obfuscation techniques [25]

These characteristics enable efficient malware triaging, especially inter- and intra-cluster analysis of interesting malware families (Section Malware Clustering Experiment), and can be used as a complement to traditional anti-virus tools by leveraging on machine learning techniques to accurately detect malware at the end host (Section Malware Detection Experiment). Indeed, our results show that, when combined with machine learning techniques, eBOFM modeling is not only outperforming the existing malware behavior modeling techniques, but its computation times (e.g., for matching signatures or clustering) and memory consumption are vastly lower as well, thus making it a much more practical and scalable approach.

In summary, this paper makes the following contributions: We propose a unified malware behavior modelling technique (called eBOFM) that captures malicious interactions between malware and security-critical OS resources in a scalable manner and at an adequate level of abstraction. When combined with machine learning techniques, eBOFM models can be directly used for automated malware detection and clustering tasks without any transformations.

We conducted and report three experiments involving two malware datasets containing 4,853 and 151,342 malware samples each. Our approach, when combined with appropriate machine learning classifiers, can achieve a malware detection rate of 91.7% with no false positives, the latter being important in our context. With respect to clustering, it can achieve an average accuracy (F1-score) of 93.9% and 62.2% on the small and large malware datasets, respectively, the latter lower score being due to a lower recall.

We demonstrate the effectiveness of eBOFM features in malware detection and clustering operations by showing that eBOFM models dramatically reduce the malware feature spaces and yet improve the detection and clustering accuracies compared to four other malware behavior modelling techniques proposed in the literature.

In addition, we also show that eBOFM models achieve a significant reduction in time (68%) and memory (72%) requirements against a well-known open-source malware clustering framework, Malheur [11], when clustering the large dataset.

2.2 Methodology

In our previous work [10], we proposed a Bounded Feature space behavior Modeling (BOFM) technique to represent malware behavior, where BOFM was originally proposed for automated malware detection. Unfortunately, BOFM features lack information that provides valuable insight into actual malicious behavior. To reduce the overhead of analyzing every single malware sample received, analysts use clustering to group them based on their malicious behavior and pick few representative samples from each group to analyze and build effective detection mechanisms [26]. Thus, extracted malware features need to be very informative but not overwhelming for the analyst. To cater for these requirements, we extend the original BOFM technique by annotating the features with security-critical attributes to be more expressive. For example, given a malware that deletes a file during its execution, using BOFM features, we are only able to represent this behavior as malware deleted a file. In contrast, with eBOFM features, we are able to infer the file type (e.g., exe, dll) and its location (e.g., system directory). This additional information provides deeper insight into malicious behavior and is considered security-critical. In addition, it enables malware analysts to prioritize malware behavior based on their severity, e.g., malicious events related to system files are given higher importance compared to the rest.

2.2.1 Operating System (OS) Resources

In general, any software program interacts with the host operating system (OS) to carry out its operations. Malware is not different and interacts with the host OS to achieve its malicious objectives. Hence, in eBOFM, we model these interactions in

a scalable manner. A typical operating system consists of various components such as file system and registry. Thus, in eBOFM we monitor how a malware interacts with these various components (OS resources) to fulfill its objectives. Based on the classification in [27] and [28], we considered nine types of OS resources to extract the run-time behavior of malware samples. They are as follows:

- **File system.** The operating system and the programs that run on it are made up of individual files.
- **Registry.** A system-defined database in which applications and system components store and retrieve configuration data⁵.
- **Process/Thread.** A process provides the resources needed to execute a program and a thread is an entity within a process that can be scheduled for execution.
- **Windows Services.** Special processes start at the system start-up time and are not tied to any interactive user [28]
- **Mutex.** They protect shared resources from simultaneous access by multiple threads or processes⁶.
- **Network.** This corresponds to network-related activities of the program being executed.
- **Host/User Information.** This represents environment specific information such as debugger and user info and privileges obtained by the running malware.
- **Desktop.** A logical display surface that contains windows, menus, and hooks.
- **Component Object Model (COM).** An interface standard that allows different software components to call each other without knowledge of their internal specifics [28]

⁵Registry (MSDN): <https://bit.ly/2O4rqG1>

⁶Mutex Objects (MSDN):<https://bit.ly/2Q03kcY>

CreateKey	DeleteKey	QueryValue
DeleteValue	OpenKey	EnumValue
SetValue	EnumKey	NotifyChangeKey

Table 2.1: Possible actions that can be performed on **Registry** resource

Analysis of malware behavior is usually carried out through examining the actions a program performs on security-critical OS resources. An action corresponds to a higher-level operation, such as reading a file, that is composed of a set of related system calls [5, 8] . For example, reading a file may require two system calls: (1) **NtOpenFile** to open the file, and (2) **NtReadFile** to read the file content. The main advantage of using actions over system calls is that different versions of the same operating system (e.g., Windows 2000 and Windows XP) may use different names for system calls that are in fact serving the same purpose [9] and, as a result, analyzing system calls directly may result in dealing with unnecessarily large amounts of data. For example, in Windows OS, the system calls **NtCreateProcess** and **NtCreateProcessEx** are both used to create a new process. Thus, these two system calls can be mapped to a single action called **CreateProcess**. System call sequences can be mapped to actions using a mapping algorithm [8] . There are a number of different mapping algorithms used in the malware research community. Depending on the algorithm used, the system call sequence $\langle \text{NtOpenFile}, \text{NtReadFile} \rangle$ can be mapped to two distinct actions **OpenFile** and **ReadFile**, respectively, or both system calls can be combined to represent a single action **ReadFile**. For example, the set of operations can be performed on a **Registry** resource can be summarized by the high-level actions shown in Table 2.1.

In eBOFM, malicious behavior is modeled based on the sets of actions, performed by malware, on individual OS resource objects. An *OS resource object* (in short, *OS object*) corresponds to an instance of an OS resource type. For example, for File resource, individual files (e.g., `C:\windows\system32\kernel32.dll`) are its

objects and they give valuable information to malware analysts. In the aforementioned example, given the file object name, malware analysts can infer the type (`.dll`) and the location (`C:\windows\system32\` or *system directory*) of the file, and decide whether the actions performed by malware are critical and need more attention. Hence, we incorporate these vital details into eBOFM features.

2.2.2 Malware Feature Representation

In eBOFM, for each OS resource type (e.g., `File`), the set of actions performed by malware (e.g., `DeleteFile`) on an OS object (e.g., `kernel32.dll`), along with its attributes, constitutes a malware feature ($\mathcal{F}_{mal}$), and it is represented by a 2-tuple as follows:

$$\mathcal{F}_{mal} := (\text{action-set}, \{\text{attr}_x\})$$

Here, `action-set` refers to the set of actions performed by malware on OS objects and `attrx` (i.e., attributes) refers to the security-critical information extracted from these OS objects that are useful to malware analysts. Here, x denotes the maximum number of attributes that can be extracted from an OS object. Based on our preliminary study, we find that no more than three attributes are required to represent a feature (i.e., $x \leq 3$). In fact, in most cases, except for `Network` resource, it is less than three. For instance, in features related to `File` resource, each `action-set` is annotated with only two attributes, namely *file type* and *location*. Further, it is also noted that for certain OS resource types, the attributes can be highly volatile and machine specific, hence, they are ignored. For example, for features involving system time, no attributes are extracted as it is specific to the machine in which the malware is running and thus, cannot be generalized. Next, we briefly explain the attributes extracted for each OS resource type.

2.2.3 Security-critical Attributes

From File objects, we extract two attributes namely, *file type* and *location*.

File type := .exe | .sys | .dll | ... | .js | .reg | others

File location := sys-dir | shared-folder | others

Here, we considered 15 possible file types, including .exe, .sys, .dll and .reg, that are commonly accessed by malware, and if a file doesn't fall under any of these 15 types, it is labelled as **others**. These file types are generally considered suspicious and thus, catch the attention of malware analysts. However, this list can be easily extended to capture any file types that are of malware analysts' interest. For file location, we check whether the file belongs to a system directory (**sys-dir**) or shared folders on a network (**shared-folder**). If its neither of those, we mark the file location as **others**. In Windows, system directory generally refers to C:\windows\system32.

For Registry resource, we looked into five key types that are commonly be abused by malware to stay persistent.

Key type := BootExecute | Service | Winlogon | Run | AppInit_DLLs | others

When the computer powers up, the Windows session manager subsystem (**smss.exe**) launches any program listed under BootExecute key (**HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\BootExecute**) and thus, if a malware manages to register itself here, it will be automatically launched at the system boot up time. On the other hand, malware may register itself as a service under Service key (e.g., **HKLM\System\CurrentControlSet\Services**) and in return, enables itself to be loaded at system startup or on demand or when triggered by some events. This gives more flexibility to the malware. Winlogon keys (e.g., **HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Winlogon\Notify**) handle the user interface, user account login and other events such as password expiry warnings, system logoff and shutdown. Hence, the malware can add itself under these keys to get

launched immediately after the user logs into the system or any event handled by Winlogon key is triggered. Further, certain applications register themselves under Run keys (e.g., `HKLM\Software\Microsoft\Windows\CurrentVersion\RunOnce`) to launch automatically each time the user logs in. This enables malware to abuse run keys to launch itself without the users knowledge each time she logs in. Next, all the DLLs specified under AppInit_DLLs key (`HKLM\Software\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs`) are loaded into every process that loads User32.dll and thus, malware too can add its DLLs to AppInit_DLLs. Finally, all other registry keys that do not fall under any of the above mentioned five types are categorized as others.

For **Process** resource, we look for its location and the process creation flag as they are helpful to determine whether the process is suspicious and need more attention.

Process location := **sys-dir** | **others**

Process creation flag := **suspended** | **others**

In this study, we limited the process locations to system (**sys-dir**) and non-system directories (**others**). However, it can be easily extended to other directories such as *temporary internet file folders*. In addition, we also monitor whether the process is created in suspend state, where, in general, malware tend to create legitimate process (e.g., `lsass.exe` or `svchost.exe`) in suspended mode and inject itself into it (a.k.a., *process hollowing*) to hide amongst the normal processes and to avoid the risk of crashing a running process through attempting process injection, e.g., *Stuxnet* [29].

For **Component Object Model**, we monitor the COM server types used by malware.

COM server type := **in-process** | **out-of-process**

The **in-process** server (implemented as DLL) runs in the process space of the calling application (e.g., malware). Similarly, the **out-of-process** server (implemented as `.exe` file) can reside either on a local computer or remote computer and

runs in its own process⁷.

For `Windows Service` resource, we look into two aspects: `service type` and `start type`.

`Service type := shared_process | own_process | kernel_driver`

`Start type := kernel | auto_load | manual_load | disabled`

Service type reveals whether the service runs in a shared process with other Win32 services (DLL), runs in its own process (EXE) or loaded as a kernel driver. Similarly, start type helps to understand whether the service created by the malware; loaded at kernel initialization, loaded automatically at system start-up, manually loaded via Service Control Manager (SCM) or disabled.

Under `Desktop`, we look for the Windows message hook types.

`Hook type := mouse_hook | keyboard_hook | others`

The common message hooks used by malware are `WH_KEYBOARD`, `WH_KEYBOARD_LL` and `WH_MOUSE`, where these hooks are used to log the keystrokes and mouse movements [30] (e.g., *TrojanSpy:Win32/Keylogger.X*⁸)

Regarding `Host/User`, we monitor the type of impersonation commonly used by the malware.

`Impersonation type := System | Administrator`

Malware, in general, try to impersonate the Administrator or System account (in most cases) to successfully execute its malicious payloads.

⁷COM (MSDN): <https://bit.ly/2PYfkM6>

⁸Key logger Trojan (2011): <https://bit.ly/2MWXjeY>

Finally, for **Network** resource, we focus on the connection type, application-level protocol and HTTP request method used by the malware.

Connection type := TCP | UDP | ICMP | others

Protocol := HTTP | SMTP | IRC | others

HTTP request method := POST | GET

HTTP, SMTP and IRC are widely used protocols by malware [Bayer et al. 2009] and it is observed that SMTP is generally used to send spam, while HTTP and IRC are mostly used for Command & Control (C&C) communications.

It is observed that **File**, **Mutex**, **Process** and **Service** names are often randomly generated by the malware during its execution, while URL (Uniform Resource Locator) and IP address are often get blacklisted and therefore, there is no agreed-upon systematic approach to generalize these highly volatile artifacts [4] . Thus, these details are abstracted away from eBOFM features, to help reduce the noise level (see the example below). Next, we briefly explain the properties of eBOFM features.

2.2.4 Properties of eBOFM features

To remain resilient to obfuscation techniques, eBOFM features hold the following four key properties:

1. Regardless of the number of times an action is performed by a malware on an OS object, it is reflected only once in the corresponding eBOFM feature.
2. The sequence in which the actions are performed by malware and the ordering of actions in an action set are ignored in feature construction.
3. Identical action sets which are performed on two different OS objects of same type, with identical attribute values are modelled as a single feature.
4. Highly volatile OS object details are not considered as attributes.

The rationale behind our modeling technique is that obfuscation techniques, such as dead-code injection, subroutine reordering, instruction substitution and code transposition, are widely used to evade traditional, signature-based malware detection [25]. It is observed that malware authors often: (1) reorder independent actions⁹ [31], and (2) repeat certain actions many times [32], e.g., perform read action in a loop to break the byte sequence (or code pattern) without affecting the semantics of the program, which will defeat byte (or action) sequence-based malware detection techniques

Therefore, accounting for the sequence of actions in behavior modeling may have the adverse effect of failing to capture identical, high-level behaviors with different action sequences. In addition, as opposed to sets, sequences of actions, similar to n-gram, would result in a huge feature space and thus, would be much less scalable. Thus, we modeled the malware behavior as a set of actions, in contrast to sequences, to overcome the above mentioned obfuscation techniques. The advantage of using sets in malware behavior modeling is twofold: (1) repeated actions are ignored, and (2) it is agnostic to the reordering of independent actions. Though it is noted that there is a trade-off in using sets as the ordering of actions may constitute valuable information, in practice, malware authors often use obfuscation techniques that render this information useless. Further, OS object details that are subject to randomness are not considered as part of the malware features, where there is no agreed-upon mechanism to generalize these highly volatile artifacts including file names, mutex values, and IP addresses [4].

2.2.5 Example

Consider the following hypothetical malware execution trace:

```
1: ReadFile ("D:\foo.txt")
2: ReadFile ("C:\Windows\system32\url.dll")
```

⁹These actions are independent from others and any permutation of these actions will lead to the same end behavior [31]

Figure 2.1: BOFM and eBOFM features extracted for the sample malware trace shown in listing:5

```
3: DeleteFile ("D:\foo.txt")
4: DeleteFile ("C:\Windows\system32\url.dll")
```

Listing 2.1: Sample malware execution trace

The above malware performs read, delete actions on a file instance `D:\foo.txt`, and performs the same set of actions on another file instance `C:\windows\system32\url.dll`. For this malware behavior, the features extracted using BOFM and eBOFM techniques are shown in Fig 2.1

2.2.6 Theoretical Upper-bound Calculation

Let us assume there are n types of OS resources and for each resource type j ($1 \leq j \leq n$), the possible actions that a malware can perform are always predefined and fixed. As a representative example, the list of actions considered for **Registry** resource is given in Table 2.1. Hence, the total number k_j of possible actions that a malware can perform on an OS object of type j is a constant. Consequently, the maximum number of possible features with regard to OS resource type j is also a constant and can be computed as follows:

$$m_j = \left[\binom{k_j}{1} + \binom{k_j}{2} + \dots + \binom{k_j}{k_j-1} + \binom{k_j}{k_j} \right] \times \underbrace{\prod_{i=1}^x a_j^i}_0 \quad (2.1)$$

where, $\binom{k_j}{h} = \frac{k_j!}{h!(k_j-h)!}$ ($1 \leq h \leq k_j$) and a_j^i ($1 \leq i \leq x$) represents the number of possible values that the i^{th} attribute of OS resource type can take. For example, for **Registry**, only one type of attribute (attr_1) is extracted and it can take 6 different values and thus, in Eq. 2.1, it is represented as $\prod_{i=1}^1 a_{Reg}^i = 6$. However,

for OS resource types where no attributes are extracted, the part denoted by O in Eq. 2.1 is replaced with the value 1. Therefore, as a result of applying eBOFM, the maximum number of possible features $\mathcal{N}$ for all resource types is always the following constant:

$$\mathcal{N} = \sum_{i=1}^n m_j \quad (2.2)$$

From Eq. 2.2, it can be seen that the eBOFM feature space does not depends on the total number of malware samples under evaluation but rather depends exclusively on the number of OS resource types, the set of actions performed, and the number of attributes extracted.

Based on Eq. 2.1, the maximum possible number of malware features that can be extracted from **Registry** is calculated as: $\left[\binom{9}{1} + \binom{9}{2} + \dots + \binom{9}{8} + \binom{9}{9} \right] \times 6 = 3,066$. In a similar fashion, for all other OS resources, the maximum number of eBOFM features is calculated. Finally, using Eq. 2.2, the total number of eBOFM features $\mathcal{N}$ that can be extracted from these nine OS resources is calculated and its sums up to 260,924 (at most 18 bits (2^{18}) are required to represent the entire feature space). Its noted that this is a theoretical maximum and, in practice, the total number of features extracted using eBOFM is much smaller than the theoretical maximum. This is due to the fact that combinations of actions (i.e., **action-set**) created using equation 2.1 can sometimes be practically infeasible and thus, cannot happen in real. The value of $\mathcal{N}$, in practice, can vary based on the number of OS resource types, list of actions considered and the attributes we want to extract in a given context. Hence, in contrast to existing approaches in which the feature space grows according to number of malware samples examined, eBOFM has a bounded feature space and this is expected to improve the scalability of malware behavior modeling. For each malware, extracted features are represented by a feature vector and we present next the feature vector construction.

2.3 Construction of Feature Vectors

In this section, we explain how we extract malware features from execution traces and embed them in feature vectors.

2.3.1 Collecting Execution Traces

The run-time behavior of malware instances is monitored using a Sandbox, which is a dynamic malware analysis tool such as CWSandbox [33], Anubis¹⁰, and Cuckoo Sandbox¹¹. These systems execute programs in a controlled environment, monitor their behavior, and generate behavior reports. These reports generally contain high-level, action based malware behavioral characteristics [33] such as newly created/modified/deleted file details, registry keys, and network traffic details. Few sandboxes, such as Cuckoo sandbox, also provide low-level behavioural characteristics (i.e., Win32 and native API calls invoked by the malware). In such cases, system calls can be mapped to relevant high-level actions using an appropriate mapping algorithm [8]. It is also noted that system calls can be used to model malware behavior but, as mentioned earlier, one must be aware of different system calls serving the same purpose (e.g., `NtCreateProcess` and `NtCreateProcessEx`).

2.3.2 Extraction of Features

Feature extraction from an execution trace involves four steps:

Step 1: OS objects present in the execution traces are identified

Step 2: Actions corresponding to each OS object are grouped and relevant attributes (if any) are extracted to form a malware feature

Step 3: Repeat *step 2* until all the OS objects identified in *step 1* are covered

¹⁰Anubis (online): Anubis (online): <http://anubis.seclab.tuwien.ac.at>

¹¹Cuckoo Sandbox (online): <https://cuckoosandbox.org/>

<div>Features</div> <div>Samples</div>	F_1	...	<div>{(read, delete)}</div> <div>{dll, SysDir}</div>	...	<div>{(read, delete)}</div> <div>{txt, !SysDir}</div>	F_N
Malware	0	0	1	0	1	0

Figure 2.2: Sample malware feature vector

Step 4: Duplicate malware features are removed

Similar to other approaches [11, 17], we use feature vectors to embed the extracted malware features. Next, we shall explain feature vector construction.

2.3.3 Embedding Malware Features

Malware features are embedded in an $\mathcal{N}$ -component (or $\mathcal{N}$ -dimensional) binary feature vector, where $\mathcal{N}$ is the total number of possible features (Eq. (2.2)). Each component in the feature vector is designated to represent a possible malware feature, where an active feature (i.e., feature present in the malware) is represented by 1 and non-existing features are represented by 0. A sample feature vector is shown in Fig. 2.2, where the n th component represents feature n (denoted by F_n). In this feature vector, we represent the eBOFM features extracted from the example. Hence, only the components corresponding to those two features are represented by 1 while the others are set to 0. It is also noted that extracting features from benign programs (used for malware detection experiments) exactly follows the same process as explained above.

2.4 Detection Method

In our study, to evaluate the effectiveness of eBOFM features in malware detection, machine learning classifiers are used to classify unknown software as either malware or benign. We tried and compared a number of ML classification techniques and report here only the results with Support Vector Machine (SVM) [34]

, which has shown to work well in a number of applications, is based on a non-probabilistic theory about learning structures in data, and yields the best results in our particular context [10]

The malware detection modeling process includes two phases: model building and model evaluation. During the model building phase, a training-set of benign and malware feature vectors is used by the ML algorithm to build a classifier. The model evaluation phase aims at assessing the classifier accuracy in a realistic fashion. By analyzing the actual classification (benign or malware) of feature vectors present in the training-set, an ML algorithm generates a trained classifier. Next, during the model evaluation phase, a test set, composed of benign and malware feature vectors, is classified by the trained classifier. Based on classification results, the performance of the classifier is evaluated by computing standard accuracy evaluation criteria. Note that computing such criteria requires the actual class labels of feature vectors in the test-set in order to compare the actual class with the class predicted by the trained classifier [35]

2.5 Clustering Method

As discussed earlier, malware triaging, especially clustering, is done at the back-end to help the analysts to understand malicious behavior. To this end, in the literature, locality sensitive hashing (called, LSH) and prototype-based clustering (called, PBCA) techniques were widely adopted by various malware clustering tools [9, 11, 17, 19, 24, 36, 37] to help cluster large number of malware samples with huge feature (or dimensional) spaces. However, it is observed that the approximation in LSH is opaque and thus provides less useful information regarding the malicious behavior to the analysts and, in the worst case, may miss some of the important malware features due to imprecision in the hashing technique. Further, it has been shown that PBCA performed well against LSH in assigning class labels to malware samples [11]

Hence, in this study, to evaluate the effectiveness of eBOFM models in malware clustering, we use the prototype-based clustering technique along with classic k-means clustering, more specifically the mini-batch k-means algorithm (called MBKM) [38], an optimized k-means algorithm well suited for large datasets. Since the k-means algorithm is well understood, we briefly describe PBCA below (for more details on MBKM, we refer the reader to [38]). In PBCA, firstly, suitable prototypes are selected, where prototypes are a subset of malware samples that are as representative of the entire set as possible, given the size of the subset. The selected prototypes are then clustered using the hierarchical clustering algorithm. Finally, labels are propagated, where un-clustered malware samples are assigned to the nearest prototype and take the label of their assigned prototype. For more details on PBCA, we refer the readers to [11]. It is important to note that this study is only conducted to evaluate the effectiveness of eBOFM modeling in malware clustering and not the clustering algorithm. Hence, in future studies, PBCA and MBKM can be replaced with any scalable, state-of-the-art clustering technique.

2.6 Experimental Design

To evaluate the effectiveness of eBOFM in distinguishing between malicious and benign behaviors, and in assigning class labels to unknown malware samples, we performed a series of experiments. In the following sub-sections, we describe the datasets and the evaluation criteria used in our experiments.

2.6.1 Experimental dataset

We used two malware datasets in our experiments. The first dataset (called Mal-large) consists of 151,342 malware samples collected from various sources such as Malheur [11], the Vxheaven malware database and honeypot. The second dataset (called Mal-small) consists of 4,853 malware samples randomly picked

from Mal-large. In this study, Mal-small is used in both malware detection and clustering experiments, whereas Mal-large is used to evaluate the scalability of our eBOFM modeling technique in malware clustering. The runtime execution traces of malware samples are collected using the Cuckoo sandbox, a well-known open-source malware sandbox.

Beside malware datasets, we need a benign dataset (called Benign) to build classifiers and perform malware detection experiments. The execution traces of 100 benign samples collected from five different real-world machines (PCs), where each machine ran 20 benign programs that are widely used in real-world settings. In addition, to lower the chances of machine specific artifacts influencing the final outcome, we also executed the benign samples in the cuckoo sandbox setup that is used to collect the malware execution traces.

The benign dataset consists of applications that are commonly used by a standard user (e.g., MS Word, Firefox) and that are mostly interactive applications. Thus, following the approach in [7], we obtained representative execution traces by simulating the user interaction with the application for around 15-20 minutes. For example, for a word processor application (i.e., MS Word), we created a new document including text, images and diagrams and saved it to disk, used several MS word plug-ins, such as EndNote, and opened few already existing documents. Similarly, for a web browser (e.g., Firefox), we connected to our university webpage and downloaded few documents from the site, uploaded files to the server, sent an e-mail using Gmail, and used several browser plug-ins such as adBlock and PDF Viewer. Likewise, for all interactive benign applications, we performed a set of common operations to get representative execution traces.

In addition, to ascertain whether and explain why a 15-minute simulated user interaction (with a benign application) is good enough to represent real-world user behavior in terms of eBOFM features, we conducted a simple experiment. We compared a representative execution trace of the MS Word application (monitored for eight hours) obtained from a real-world machine with an execution trace of the

Table 2.2: Comparison of execution traces for simulated user and real-world user interactions

	Real-world User Interaction (A)	Simulated User Interaction (B)
Application	MS Word	MS Word
Execution environment	Real-world PC	Real-world PC
Execution time	8 hours	15 minutes
Size of execution trace	129,108 KB	27,313 KB
# of system calls obtained	955,463	180,614
Extracted eBOFM features	124	192
# of common features	107 (86.3%)	107 (55.7%)

same application but with simulated user interaction. The findings are summarized in Table. 2.2.

From Table 2.2, it can be seen that the execution trace obtained using real-world user interactions (trace A) is larger in size and has a larger number of system calls than the execution trace with simulated user interaction (trace B). However, the number of features extracted using eBOFM is larger for trace B than for trace A. This shows that in practice, people often tend to use a small fraction of operations (e.g., intuitive operations) and thus generate large execution traces with limited number of useful features. This lends credence to our notion that simulated executions seem to be an effective means of acquiring traces for benign programs.

When selecting the benign applications for evaluation, we made sure that they were from one of the following functionally diverse categories: word processors (e.g., MS Word), text editors (e.g., MS WordPad), command-line shell (e.g., cmd.exe), web browser (e.g., Firefox), file transfer (e.g., Filezilla FTP server), remote access (e.g., Putty SSH client), e-mail (e.g., MS Outlook), IDE (e.g., MS Visual Studio), media (e.g., VLC player), game (e.g., Chess Titans), anti-virus tool (e.g., AVG antivirus), VOIP (e.g., Skype), cloud storage (e.g., Dropbox), reader (e.g., Adobe PDF reader) and other utility tools (e.g., Google Desktop). We observed that execution traces of benign applications in the same category looked similar. For example, using eBOFM, we managed to extract 113 features from Notepad and 131 features from WordPad execution traces, out of which 98 features were common

to both applications. That is, 86.7% of Notepad features and 74.8% of WordPad features are identical. This indicates that, as long as we include a few benign applications from each category, it is sufficient to cover a wide range of benign applications in terms of functionality. Thus, we can confidently say that our benign dataset of 100 applications (at least three from each category) is sufficiently representative to build classifiers to accurately predict malware, as confirmed by our experimental results.

Finally, malware detection consists of two phases: model building and model evaluation. Hence, we used the Mal-small to train the classifiers while using a different dataset to test them (called Mal-test), containing 300 malware samples obtained from the VirusShare database. Similarly, we used the execution traces of benign samples obtained from 4 (out of 5) machines and the corresponding benign traces collected from the cuckoo sandbox (in total, 160 benign traces) as training set (called Benign-train), while the benign traces obtained from the 5th machine and the corresponding benign cuckoo sandbox traces (in total, 40 benign traces) are used as test set (called Benign-test). This process is repeated five times where, on each iteration, the benign test set (i.e., Benign-test) is obtained from a machine that was not used in the previous iterations along with the corresponding benign cuckoo sandbox traces. Finally, averages across the five experiments are reported in the paper.

2.6.2 Evaluation Criteria

[5] have evaluated 200 different malware behavior modeling techniques, among which, the 2-bags of 2-tuples of actions with arguments model yields better malware detection accuracy (99%) compared to the rest, with a relatively low false positive rates (0.4%). Similarly, Malheur [11], a well-known open-source malware clustering framework, uses MIST behaviour models that generates scalable malware features. In addition, several other malware clustering and triaging frameworks rely on the classical n -gram models [5, 24, 39].

Therefore, in this study, we evaluate the performance of eBOFM features, both in malware clustering and detection experiments, against the following three sequence-based behavior modelling techniques: (1) 2-bags of 2-tuples of actions with arguments (aka, bag-of-tuples) [5], (2) MIST model [11], and (3) bigram of actions with arguments (aka, bigram)[39]. In addition, we also compare our previous modelling technique, BOFM [10]. Finally, it is worth noting that based on a small scale experiment with 200 malware samples, randomly selected from our dataset, we find that for n-gram models, bigram yields better malware clustering and detection accuracy. Hence, other variants of bigram (e.g., trigram) models are not included in the final comparison.

2.6.3 Experiment Termination Criteria

In this study, to remain within a feasible computing budget, we limit the extracted malware features to 10 million, which is nevertheless much larger than other experiments in the literature. That is, during the feature extraction process, when the number of malware features reaches 10 million, we stop extracting more features from the dataset. In addition, we upper bounded the time and memory usage such that an experiment (malware detection or clustering) is forcefully terminated if it executes for more than 5 hours or consumes more than 20GB of memory, whichever comes first. That is, if an experiment runs for 5 hours and it is not *naturally* terminated, we kill the process by raising *time out error*. Similarly, if an experiment consumes more than 20GB of memory, we terminate it by raising *memory out error*.

These constraints are meant to be realistic and help to evaluate the practicality of the proposed behaviour modelling technique. For example, having more than 10 million malware features may lead to huge memory (e.g., > 20GB of memory) and/or execution time (e.g., > 5 hours), hence making the modelling technique unsuitable for malware detection in practice. Similarly, a huge feature space is not helpful for malware analysts who go through them manually to generate efficient detection mechanisms.

In malware clustering experiments, the most precise way to measure clustering accuracy is to assess the degree of similarity between the ground truth and the clusters generated by the proposed clustering framework. Unfortunately, no such ground truth exists for malware samples [9, 40]. Thus, following the approaches in [9] and [11], we constructed the ground truth clusters for Mal-small using malware class labels assigned by six commercial anti-malware vendors (e.g., F-Secure and Symantec). However, for Mal-large, we constructed the ground truth clusters by relying only on one anti-malware vendor, Kaspersky, as it is so large that it would entail an unrealistically high amount of manual effort to analyze class labels assigned by each anti-malware vendor and remove the malware samples that are inconclusive (i.e., not flagged as malicious by all six vendors). Finally, these ground truth class labels are then used as a comparison baseline to evaluate the class labels assigned by the proposed clustering framework.

For Mal-small, we collected the malware class labels from Virustotal¹² for the following six commercial anti-malware products: F-Secure¹³, Ikarus¹⁴, Symantec¹⁵, Kaspersky¹⁶, VirusBuster¹⁷ and AVG¹⁸. In constructing the ground truth clusters, following the approaches in [8], [10] and [11], we only considered the malware family name and ignored the variant name.

In Table 2.3, we summarize, for both Mal-small and Mal-large, the number of clusters generated by each antivirus product. From Table 2.3, we can see that there are clear inconsistencies across antivirus products in assigning labels to these malware samples. As reported by Bailey et al. [13], this is a well-known problem among commercial antivirus products. Note that this study does not aim at addressing these inconsistencies but instead compares the clusters generated by the proposed malware behavior modeling technique against each of these ground

¹²Virustotal: <https://www.virustotal.com>

¹³F-secure: <https://bit.ly/2lar6ov>

¹⁴Ikarus: <https://bit.ly/2I8h5Uc>

¹⁵Symantec: <https://symc.ly/1Bj8Ooa>

¹⁶Kaspersky: <https://bit.ly/2gUrG99>

¹⁷VirusBuster: <https://bit.ly/2NxqpXC>

¹⁸AVG: <https://bit.ly/2McV2AN>

Table 2.3: Summary of ground truth clusters. Number of clusters generated based on the malware labels assigned by each AV vendor

Dataset	Ikarus	AVG	Kaspersky	F-secure	Symantec	Virusbuster
Mal-small	461	503	429	392	415	486
Mal-large	-	-	6,789	-	-	-

truth clusters, and quantify the differences using metrics explained in the next section.

2.6.4 Evaluation Measures

For malware detection, we employ standard evaluation measures as described in [35]. The contingency table above summarizes the four possible outcomes (i.e., TP, FP, FN and TN) from a binary classifier. True Positive Rate (TPR) is in our context the proportion of malware samples correctly classified as malware. Similarly, False Positive Rate (FPR) is the proportion of benign samples misclassified as malware. Finally, Total Accuracy (TA) measures the overall proportion of correctly classified instances, either malware or benign.

For malware clustering, we follow the approaches in [9, 11, 40], to assess clustering accuracy. We employ three standard evaluation metrics: precision, recall and F1-score, where F1-score combines both precision and recall. Here, precision evaluates how well an algorithm assigns samples of different families to their respective cluster and recall evaluates to which extent an algorithm consistently assigns samples of the same type to the same cluster.

2.6.5 Experimental Setup

All the experiments were executed using a 6 core Xeon(R) with 24GB RAM machine installed with Ubuntu 12.04. In addition, a popular machine learning (ML)

toolbox, Sklearn¹⁹, was used for ML functionalities in our implementations. Finally, we obtained the PBCA implementation from the publicly available Malheur tool²⁰.

2.7 Experimental Results

In this section, we report the results generated by eBOFM models for both malware detection and clustering experiments. In addition, we also compare the performance of eBOFM models against the selected four other malware behavior modeling techniques in terms of these three criteria: (1) malware detection and clustering accuracy, (2) scalability of the generated malware feature space, and (3) time and memory requirements.

2.7.1 Malware Detection Experiment

To assess the accuracy of our classifiers in detecting malicious code, we measured the false positive rate and total accuracy achieved by all five malware behavior modeling techniques with Support Vector Machine (SVM). As explained earlier, we used Mal-small and Benign-train to train the classifiers and Mal-test and Benign-test to evaluate their performance in terms of malware detection accuracy. The experiments are repeated five times with different benign test sets, following a standard 5-fold cross validation process.

The average results, across all five experiments using five different malware behavior models, are presented in Table. 2.4, reporting both rates and counts. From Table. 2.4, it can be seen that eBOFM models achieve significantly better detection accuracy, with no false positives. In malware detection, a lower (or zero) false positive rate is strongly desired since the consequences of flagging benign applications as malware can be expensive or even disastrous. For example, if a benign

¹⁹Scikit-learn: <http://scikit-learn.org/>

²⁰<https://github.com/riECK/malheur>

Table 2.4: Summary of Malware Detection Accuracy achieved by SVM.

Model	False Positive Rate	Total Accuracy	Counts
Bag-of-tuples	0.045	0.794	1262/1600
MIST	0.055	0.721	1154/1600
Bigram	0.060	0.693	1109/1600
BOFM	0.015	0.831	1330/1600
eBOFM	0.000	0.917	1467/1600

system file is flagged as malicious and deleted from the system, in the worst case, it may prevent the system from booting.

There is also a considerable improvement in the detection accuracy achieved by eBOFM models (91.7%) over BOFM models (83.1%). This suggests that expressive features (i.e., features containing more information about the malicious behavior) perform significantly better in terms of detection accuracy, over less expressive features. However, the additional information needs to be meaningful not to increase the noise level among the features and negatively impact detection accuracy. For example, even though bigram and MIST models are of the same type, we can see that bigram models (i.e., bigram of actions with arguments) perform poorly compare to MIST models (i.e., bigram of actions with abstracted arguments) due to low signal-to-noise ratio among the bigram features. Hence, we can confidently say that malware features that contain meaningful information about the malicious behavior are likely to detect malware with better detection accuracy compared to less meaningful features.

It is also observed that for bag-of-tuples models, the standard deviation (SD) in detection accuracy, across all five experiments, is relatively high (3.79%) compared to the other models (avg. SD 0.84%). Because of the limitation on the number of malware features (limited to 10 million), SVM couldnt learn a sound classifier model for bag-of-tuples, which lead to high standard deviation. Further, when analyzing the malware samples undetected by eBOFM, we find that few of them could not execute beyond a point as they were waiting for some instructions from their command & control (C&C) servers (probably taken down at the time

of analysis) and several others simply quit executing due to unfulfilled system requirements and anti-sandboxing techniques employed by the malware authors.

2.7.2 Malware Clustering Experiment

We compared the clusters generated by eBOFM models with the ground truth clusters and did the same for clusters generated by the remaining four malware behavior modeling techniques proposed in the literature. Tables 2.5 and 2.6 summarize the experiment results obtained for Mal-small using PBCA and MBKM clustering algorithms, respectively. From the results, it can be seen that eBOFM models clearly outperform the results obtained by the other modeling techniques with an average clustering accuracy of 93.9%²¹. Further, the improvement brought by eBOFM models is consistent across all six different benchmarks. A statistical test of differences in proportions (chi-squared test) [41] reveals that for both algorithms, the average improvement in clustering accuracy achieved by eBOFM models over other models is statistically significant at $\alpha = 0.01$.

Interestingly, MBKM yields slightly better clustering accuracy compared to PBCA (1% on average). However, it comes at a high performance cost and, hence, is not well suited for large-scale malware clustering (discussed in the next section). For bag-of-tuples models, MBKM algorithm could not converge due to huge feature spaces, which led to memory out errors. Unfortunately, computational complexity (i.e., execution time and memory requirements) is perhaps an even more serious problem for large datasets. For example, bag-of-tuples and bigram models are removed from large-scale malware clustering experiments due to unacceptably large feature spaces, where the upper limit of 10 million features is reached with only 4% and 43% of the malware samples analyzed from Mal-large for bag-of-tuples and bigram models, respectively. Further, for Mal-large, using MBKM, we were unable to obtain results for the eBOFM, BOFM and MIST models due to time out errors. Hence, for large-scale malware.

²¹eBOFM models achieved an average precision and recall of 98% and 90.2%, respectively

Table 2.5: Clustering accuracy achieved by PBCA for mal-small

Ground truth source	Mal-small (F1-score)				
	Bag-of-tuples	MIST	Bigram	BOFM	eBOFM
Kaspersky	0.711	0.861	0.831	0.901	0.969
Ikarus	0.727	0.847	0.827	0.882	0.959
AVG	0.702	0.827	0.825	0.867	0.942
F-Secure	0.725	0.795	0.762	0.825	0.895
Symantec	0.691	0.761	0.741	0.804	0.886
VirusBuster	0.762	0.859	0.833	0.892	0.964
Average Accuracy	0.720	0.825	0.803	0.862	0.936

Table 2.6: Clustering accuracy achieved by MBKM for mal-small

Ground truth source	Mal-small (F1-score)				
	Bag-of-tuples	MIST	Bigram	BOFM	eBOFM
Kaspersky	Memory Error	0.871	0.837	0.914	0.971
Ikarus		0.852	0.831	0.89	0.962
AVG		0.837	0.828	0.878	0.945
F-Secure		0.815	0.771	0.841	0.905
Symantec		0.771	0.749	0.818	0.899
VirusBuster		0.866	0.842	0.905	0.967
Average Accuracy		0.835	0.810	0.874	0.942

Table 2.7: Clustering accuracy achieved by PBCA for mal-large

MIST			BOFM			eBOFM		
Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
0.753	0.302	0.431	0.782	0.321	0.455	0.896	0.476	0.622

From the results it can be seen that the overall F1 scores achieved by the models under consideration are relatively low compared to the scores they achieved for Mal-small. For example, eBOFM achieved an average clustering accuracy of 93.9% for Mal-small but only 62.2% for Mal-large. The major contributing factor for this reduction in F1 score is recall (47.6%), which indicates that large malware families in the ground truth clusters are divided into several small groups. This observation is not unusual in malware clustering, especially with large datasets

[Hu et al. 2013b]²². However, precision is around 90%, which is very encouraging from the malware analysts perspective. That is, eBOFM models combined with PBCA clustered 151,342 malware samples into 8,127 groups.

Finally, we can conclude that the results reported in Tables 2.4, 2.6, 2.5 and 2.7 reveal that eBOFM models can be effectively used for both malware detection and clustering tasks.

2.7.3 Scalability of eBOFM Models

In this section, we evaluate the scalability of eBOFM models based on two factors: (1) extracted malware feature space, (2) execution time and memory usage for each experiment (i.e., malware clustering and detection). Table. 2.8 summarizes the number of malware features extracted by each behavior model under evaluation. As mentioned earlier, bag-of-tuple and bigram models are removed from large-scale analysis as the number of covered malware samples, in Mal-large, is less than 50% when the extracted features reached the upper bound of 10 million features¹⁴. From Table. 2.8, it can be seen that compared to bag-of-tuples, bigram and MIST models, eBOFM dramatically reduces the malware feature spaces while comprehensively outperforming them in both malware detection and clustering experiments. Interestingly, BOFM achieves 1.2x and 3.2x reduction in feature spaces, against eBOFM model, for Mal-small and Mal-large, respectively. However, that improvement comes at the cost of considerable reduction in malware detection and clustering accuracies, where eBOFM models achieve an average improvement of 10.2% in malware detection accuracy with 36.7% and 7.7% improvements in malware clustering accuracy for Mal-large and Mal-small, respectively. In addition, generating 13,372 malware features, without using any feature reduction technique, from a dataset with more than 150,000 malware samples is very reasonable, given that eBOFM achieves a 383x feature reduction over MIST models.

²²In [24], the authors achieved a recall of 25% and a precision of 82%, when clustering 130K samples using static malware features combined with PBCA clustering.

In Table. 2.9, we have summarized the memory requirement for the malware behavior modeling techniques under evaluation for both malware detection and clustering experiments. From the results it can be seen that eBOFM models clearly outperformed the sequence-based models. However, BOFM models show a slight edge over eBOFM models in malware clustering, where it reduces the average memory requirement by around 40% compared to eBOFM. In contrast, for malware detection, BOFM reduces the average memory requirement by only around 5% compared to eBOFM. As discussed earlier, this observation can be directly linked to the feature spaces generated by BOFM and eBOFM models. Further, it is also evident that sequence-based models' memory requirements are quite unrealistic for a malware detector that runs on the host (i.e., end users PC). For example, bag-of-tuples requires more than 13GB of memory to perform the detection operation, which is clearly undesirable and impractical.

For malware clustering tasks, MBKM requires more memory than PBCA and the difference is very significant: MBKM consumes 7x more memory than PBCA. Hence, comparing the average improvement of 1% in clustering accuracy achieved by MBKM over PBCA with the additional memory required by MBKM, one can clearly rule out the latter. One of the reasons for this observation is that MBKM tries to keep all the data in memory during the clustering process and unfortunately, malware models, in general, have huge feature spaces that lead to this unacceptably large memory requirement. PBCA performs clustering only on the prototypes (or exemplars), which are significantly less than the actual data set. For example, PBCA selected only 11,367 samples as prototypes from Mal-large with 151,342 samples (a reduction of around 92%) and performed hierarchical clustering on them to arrive at a final number of 8,127 clusters.

Finally, Table. 2.10 summarizes the time taken for each model to perform malware detection and clustering tasks. It is important to note that for malware detection tasks, only the classifier training time is presented in the table as the testing time is very short for each model (at most 0.04 second), with BOFM and eBOFM being the fastest (0.001 second) and bag-of-tuples being the slowest (0.038 second) to detect malware. As for the training time, BOFM and eBOFM managed to

Table 2.8: Summary of Malware features generated

Dataset	Malware behavior modelling techniques				
	Bag-of-tuples	MIST	Bigram	BOFM	eBOFM
Mal-small	10,000,000	283,762	868,116	592	1291
Mal-large	-	5,131,291	-	3,148	13,372

Table 2.9: Memory requirements (in Megabytes) for each malware behaviour modelling technique

Model	Malware detection	Malware clustering		
	Mal-small	Mal-small		Mal-large
	SVM	PBCA	MBKM	PBCA
Bag-of-tuples	13,718.7	1,511.4	Mem. out	-
MIST	466.2	198.7	1,287.7	8,773.4
Bigram	1,096.4	397.9	3,678.1	-
BOFM	118.3	18.6	83.3	1,869.5
eBOFM	124.9	29.1	112.7	2471.9

train the classifiers within a fraction of a second while, as expected, bag-of-tuples took around 57 seconds. On the other hand, to cluster Mal-small, MBKM takes on average 19.4x more time than PBCA. In addition, for bag-of-tuples, MBKM fails to generate any results within the allowed memory limit. Further, PBCA is most suitable, among the evaluated clustering algorithms, for large-scale malware clustering tasks, since it took roughly 74 minutes to cluster a large dataset with over 150,000 malware samples.

2.7.4 Qualitative Analysis of eBOFM Features

In this section, we take a deeper look into malware features extracted by eBOFM, where we analyze the inter- and intra-family feature semantics for selected malware classes. Our goal is to assess whether eBOFM could provide useful insights to a malware analyst regarding the features that distinguish and characterize malware

Table 2.10: Time requirements (in seconds) for each malware behaviour modelling technique

Model	Malware detection	Malware clustering		
	Mal-small	Mal-small		Mal-large
	SVM	PBCA	MBKM	PBCA
Bag-of-tuples	56.886	654.8	Mem. out	-
MIST	1.230	112.6	1,938.98	13,854.6
Bigram	4.672	279.1	6,078.27	-
BOFM	0.308	8.3	119.82	2,915.7
eBOFM	0.334	11.7	249.72	4,390.6

Figure 2.3: (a) shows the Intra-family analysis of Allapple Worm. (b) shows the Inter-family analysis between Looper Trojan, Allapple worm and AdRotator adware. Darker area represents active malware

families. To this end, we co-clustered [?] the malware samples and their features to effectively identify the distinguishing and common features present in and between malware samples from various malware families. Given a $n \times m$ matrix, where each row represents a malware feature vector and each column represents a particular malware feature, a co-clustering algorithm generates a subset of rows which exhibit similar behavior across a subset of columns, e.g., malware instances that exhibit similar active features (for more details we refer the readers to [19])

It is observed that malware analysts often look for distinctive malware behaviors to generate efficient detection mechanisms, where distinctive behavior refers to malicious behavioral patterns particular to a malware family that differentiate it from the rest. To this end, co-clustering (both inter- and intra-family) enables the malware analyst to efficiently identify the distinctive and common features present in most of the malware samples and of course, uninteresting (noise) features, hence, leading to efficient malware signature generation.

As an example, Fig. 2.3(a) depicts the co-clustered malware feature space of 287 Allapple worm samples. Dark areas represent active features (i.e., features present in the malware). Among 35 malware features, extracted from 287 Allapple worm instances, around 74.3% (26) of the features were common across all instances (labeled d2). Analyzing those common features, we observed that Allapple worm performed a few operations that are unique to its family (distinctive behavior), such as: (1) copy the malicious executable to the Windows system directory and deletes the original file; and (2) execute the malicious binary as a system service that starts automatically at system startup. These distinctive behaviors are precisely modeled by eBOFM.

For example, the feature ($\{CopyFile, DeleteFile\}, \{exe, sysDir\}$) represents the first behavior and ($\{CreateService, OpenService, ChangeServiceConfig, StartService\}, \{auto.load, own.process\}$) reflects the second behavior. Interestingly, eBOFM models also identify variants within malware families. For example, 150 out of 287 Allapple instances performed extensive ping scans in an attempt to connect to several remote locations (labeled d1 in Figure 4(a)). This behavior distinguishes Allapple.2 from Allapple.1 [9].

Similar to intra-family, inter-family co-clustering provides useful insights to the malware analyst into identifying distinctive, common and noise features across various malware families. As a representative example, Fig. 2.3(b) depicts the inter-family analysis of three malware families: Allapple worm, AdRotator adware and Looper Trojan. Looper Trojan creates and deletes several temporary files and this behavior distinguishes Looper from the other two malware families (labeled

d5). Similarly, the behavior of copying itself to the system directory and deleting the original file distinguishes Allapple from the other two malware families (labeled *d4*). Finally, around 54.3% of the features (75 out of 138) clearly distinguish AdRotator from the rest (labeled *d3*).

Compared to inter- and intra-family feature semantics generated using existing malware behavior modeling techniques, such as n -gram (refer to [19]), eBOFM models not only generate a more confined feature space but also provides deeper insight into malicious behavior. In addition, eBOFM also minimizes uninteresting features (noise) when modeling the malware behavior. This can be observed from Figures 4(a) and 4(b), where there are much fewer noise features (labeled $n1$ and $n2$) than core malware features (d labels). Noise features can be easily spotted in the figures (co-clustered feature spaces) as they show as isolated active features that are not common to malware samples within families or do not help to reliably distinguish them.

2.8 Related Work

Past research has yielded several innovative and useful techniques for malware detection and clustering. In this section we summarize the malware behavior modeling techniques proposed in the literature for both malware clustering and detection tasks.

2.8.1 Behaviour Modelling for Malware Clustering

Malheur, proposed by Rieck et al. [11], addresses several key issues present in previous malware clustering techniques [9, 12, 13, 23]. Its key contribution is prototype-based clustering, where prototypes benefit from explicit expression of malware features that can be easily inspected by security analysts, whereas locality sensitive hashing (LSH) clustering, proposed by Bayer et al [9], is opaque and does not provide useful information to the analysts [11]. However, Malheur

uses an n-gram based malware behavior modeling technique that generates huge feature spaces and for which extracted features are not robust against basic obfuscation techniques. Huge feature spaces make the malware clustering process very challenging [24] and increase the workload of security analysts who go through them to generate malware signatures. In addition, malware behavior modeling techniques based on system call sequences can be easily evaded using simple obfuscation techniques such as reordering of independent system calls. Thus, it is not a reliable approach as malware authors often employ these techniques to evade pattern-based malware detection

Besides Malheur, Bayer et al. [9] proposed a malware clustering technique that models the runtime behavior of malware using OS objects and operations in a fine-grained manner and that uses Locality Sensitive Hashing clustering. The major drawback in fine-grained behavior modeling is that the system and malware-specific details (highly volatile artifacts) introduce unnecessary features and increase the amount of noise in the malware features space. For example, a malware may create files with different names in distinct executions and there is no systematic approach to generalize these highly volatile artifacts [4]. Jang et al. [19] proposed an automatic malware triaging technique, BitShred that uses hashing technique to dramatically reduce the high dimensional feature space that is common in malware analysis. One of the major drawbacks in feature hashing is the occurrence of hash collisions, where two distinct malware features have the same hash value. In Jang et al. [19], it is reported that the error in distance calculation, due to collision, leads to misclassification of malware samples.

Other approaches such as Lee et al. [12] propose a behavior-based malware classification that compares the sequences of system calls to determine the similarity of two executables. Such a fine-grained approach makes it difficult to abstract the underlying malicious behavior and further, semi-supervised learning is not desirable as it involves training. Bailey et al. [13] modeled the malware behavior as non-transient state changes, where the state changes are extracted with a higher level of abstraction than single system calls. However, useful information obtained from these state changes is very limited.

On the other hand, several techniques proposed in the literature leverage on Malheur [11] for scalable clustering, but use various malware behavior modeling techniques [17, 24, 36, 42]. The most recent work [24] extracts n-gram features from malware binary opcode sequences. Static analysis-based modeling is efficient but has severe limitations: they are less effective when obfuscation, encryption and packing of malware binaries are in use [2]. Further, in Hu et al. [24], a hashing technique is used to reduce the huge feature space but, as mentioned above, hashing may lead to information loss due to hash collisions. The works in Hu et al. [24] and Qiao et al. [42] also use n-gram models to extract malware features and thus, suffer from the high dimensionality problem [43]. In Chandramohan et al. [17], coarse-grained malware features are extracted from execution traces but the extracted features are too abstract and not useful for malware analysts.

2.8.2 Behavior Modelling for Malware Detection

The study on behavior-based malware detection reported by Canali et al. [5] is most relevant to our current work. In Canali et al. [5], the authors have performed a comprehensive study on analyzing the efficiency of various malware behavior modeling techniques. They organized the behavior models based on three dimensions: (1) the granularity of basic model elements (i.e., system calls at various levels of abstraction, such as with and without parameters), (2) the relationship among basic model elements, such as n-gram, m-bag and k-tuple¹⁵ and, (3) cardinality of each specification (i.e., values of n, m and k). A series of more than 200 experiments were conducted on a large set of malware and benign samples to systematically identify the optimal behavior model for malware detection. Their experiment results revealed that the optimal behavior model, 2-bags of 2-tuples of action with arguments, achieved a 99% detection rate with only a 0.4% false positive rate.

However, there are several shortcomings associated with this approach. The most important issue is that the scalability of this approach is highly problematic. Behavior modeling techniques such as n-gram, m-bag and k-tuple can easily generate

huge feature spaces. For example, from an execution trace with 400 unique system calls, more than 10 million features are constructed using m-bag with a bag cardinality of 3 ($m = 3$). This problem is even worse in tuple-based behavior models. Further, it is also reported that the memory consumption is one of the major threats to the practical implementation of this approach, where on a standard machine (Intel Dual Core 2.66 GHz with 4GB of RAM) the prototype malware detector consumed 1GB of RAM for 5 million malware signatures. Last, it is also reported that feature extraction from malware samples is itself a computationally intensive task, where almost 2 days of computation are required for the tuples of system calls with arguments model

In addition, the approach in Canali et al. [5] requires the tuning of several parameters for optimal performance, which makes it less practical. Indeed, it was observed that intensive parameter tuning is often associated with overfitting problems. For example, to manage the huge feature space, authors have proposed a feature pruning mechanism. A malware feature is discarded if it is not general enough (i.e., does not detect at least five malwares) or too redundant (i.e., represented by 20,000 other signatures), where the values 5 and 20,000 are arbitrarily selected. Further, there is a trade-off in selecting the cardinality of a specification (e.g., number of basic model elements in a bag), where increasing the cardinality may result in a high detection rate but also lead to overfitting. For example, it is reported that detection rates achieved by tuple-based behavior models are highly sensitive to cardinality. Finally, the alert threshold, the number of malware signatures that need to be matched to flag an unknown program as malware, significantly influences the detection rate and false positive rate, where it is shown that a small alert threshold results in high detection and false positive rate and vice versa. Thus, choosing the appropriate values for these parameters is crucial but complex in practice.

Apart from Canali et al. [5], there are several other behavior-based malware modeling techniques proposed in the literature. Lanzi et al. [4] proposed AccessMiner, a malware detection technique based on system-centric malware models, where the

interaction between benign sample and the OS resources is modeled in a system-centric manner. This addresses the limitation of program-centric approaches. Further, authors empirically demonstrated the inefficiencies of n-gram based malware behavior modeling. In addition to generic malware detection [31, 44], behavior modeling based on system calls and library calls is also proposed to detect more specific classes of malware, such as spyware and botnets (Stinson et al. [45]).

Kolbitsch et al. [3] proposed an efficient and effective malware detection approach at the end host, where the malware behavior is modeled as a graph and detection is done at the end host using graph matching. Similarly, Fredrikson et al. [7] proposed malware specification mining using dependency graphs. They managed to achieve a higher detection rate than two commercial behavior-based malware detectors. However, graph mining still remains very computationally intensive, as it was reported that it took around 12-48 hours to extract malware specification from certain network worms. In addition, the datasets used for validating most of these approaches are very limited [3, 31, 44, 45]. For example, Kolbitsch et al. [3] used only 563 malware samples and 5 benign samples for evaluation.

In contrast to the aforementioned malware behavior modeling techniques proposed for both malware clustering and detection tasks, the work presented in this chapter achieves a better clustering and detection accuracy with a considerable performance improvement, in terms of time and memory requirements, through a unified malware behavior modelling technique.

2.9 Conclusion

In this chapter, we proposed a bounded feature space behavior modeling technique (called, eBOFM) which, when combined with appropriate machine learning and clustering techniques, can lead to effective and scalable malware detection and clustering (or triaging). The malware features extracted using eBOFM are very expressive in nature and provide valuable information to security analysts to build malware protection mechanisms. Most importantly, the eBOFM feature space is

bounded by an upper limit N , a constant. Using large-scale experiments, we show that, with eBOFM, computation time and memory usage are vastly lower than in existing detection and clustering techniques, while still improving their detection and clustering accuracy.

In the future, we plan to build an online malware detection framework based on eBOFM models, where eBOFM features are extracted on-the-fly and are compared against the learnt malware behavior models. As demonstrated in this study, the malware feature reduction along with the dramatic reduction in time and memory requirements achieved by eBOFM models are very encouraging. Therefore, we strongly believe that these advantages will enable us to perform online malware detection in a scalable and efficient manner.

3

Security Patch Analysis for Binaries - Towards Understanding the Pain and Pills

3.1 Introduction

Program vulnerability is one of the major threats to software security. However, it is almost impossible to avoid vulnerabilities at the development stage; it is even difficult to discover vulnerabilities at the production stage. Security experts usually leverage dynamic fuzzing (e.g., [46, 47]), symbolic execution (e.g., [48, 49]) or static code auditing (e.g., [50, 51]) to find vulnerabilities. However, none of these techniques can provide a complete solution to win the war against vulnerabilities. Dynamic fuzzing suffers from the code coverage problem and the initial seeds problem [52]. Symbolic execution cannot scale well to real-world programs, due

to path explosion and constraint solving problems. Static code auditing often requires human expertise, and cannot scale well when the program complexity increases.

Vulnerabilities, once discovered, are often fixed by applying security patches. In that sense, security patches carry important information about vulnerabilities. By focusing on the remedy of vulnerabilities instead of the vulnerabilities themselves, patch analysis has been proposed to discover n-day vulnerabilities, whose patches have been released but not deployed to every instance of the software in the world. Since patch analysis is relatively accurate to find vulnerabilities, it has gained the popularity in both industry and academia. Also, security patches are good entry points to understanding the program weaknesses and how the vulnerabilities work inside the program, especially for security participants who do not have access to the source code. Moreover, the underlying information of patches has been used to build automatic bug-fixing tools [53–55], generating valid patches for similar vulnerabilities. However, up till today, most existing researches on patch analysis work at the source code level [56, 57], but very few works have been done to tackle this problem at the binary-level.

Binary-level patch analysis can only be performed on machine instructions for closed source programs without symbol tables, which often requires a significant amount of human efforts and expertise to understand the semantics of instructions. Due to this complex nature, the existing techniques [58, 59] often heavily rely on manual or heavy program analysis, which becomes infeasible for real-world programs.

On the other hand, software companies may tend to patch the vulnerabilities they find themselves in a secret way instead of making them public and applying Common Vulnerabilities and Exposures (CVE) numbers due to their security regulations or policies. As a result, security analysts cannot know the existence of particular vulnerabilities, which hinders the understanding and analysis of vulnerabilities. Even worse, in some cases, only the patched binary programs are

available; while the patches themselves may not be publicly released, which hinders the existing patch analysis techniques that often rely on the availability of patches. Moreover, due to the application of patch obfuscation and patch modification techniques such as honey-patching [60], the patch patterns in open source binaries may differ greatly from close source ones.

To address these problems, we propose a scalable binary-level patch analysis framework, named SPAIN, to automatically identify security patches and summarize patch patterns and their corresponding vulnerability patterns. In particular, given the original and patched versions of a binary program, SPAIN locates the functions that have been changed from the original binary to the patched binary. Then, it detects the changed traces (i.e., a sequence of basic blocks) for each patched function to capture the function-level changes. These traces may contain security or non-security patches. Finally, it identifies security patches through a semantic analysis of these traces and summarizes the patterns through a taint analysis on the patched functions. As we run more binary programs, we can maintain a database of patterns that will be continuously enriched to embrace the evolving and emerging of various vulnerabilities. There are many impactful applications of the learned patterns in binary security like patch detection, vulnerability detection, automatic patch synthesis, and even patch/vulnerability trend analysis with the help of data analytic techniques. A more detailed discussion can be found in Section 3.3.5.

We evaluated SPAIN on several real-world projects. The experimental results demonstrated that SPAIN can identify security patches with high accuracy and high scalability, and summarize 5 security patch patterns and their corresponding vulnerability patterns for 5 types of vulnerabilities. Moreover, we discovered undocumented security patches and found 3 zero-day vulnerabilities in Adobe PDF Reader.

In summary, our work makes the following contributions.

- We proposed a scalable binary-level patch analysis framework SPAIN, which can identify security patches and summarize patch patterns and their corresponding vulnerability patterns.
- We implemented SPAIN in a prototype and conducted experiments to demonstrate the accuracy, scalability, and application of SPAIN.
- We discovered undocumented security patches and found 3 zero-day vulnerabilities in Adobe PDF Reader.

3.2 Preliminaries and Overview

3.2.1 Preliminaries

In this work, we assume that binary programs are in the X86 32bit format. Each binary program contains a number of functions with an entry point (starting function). This section introduces several basic concepts in binaries, i.e., basic block, control flow graph and partial trace.

Definition 3.1. A *basic block* in a binary function is a straight-line sequence of X86 instructions with no branches in except to the entry and no branches out except at the exit.

Definition 3.2. Given a binary function, the *control flow graph* (CFG) of the function is a tuple $G = (N, E, N_s, N_t, \iota)$, where N is a finite set of nodes, and each node represents a basic block in the function, $E : N \times N$ is a set of edges that connect two nodes and represent the control flow from one basic block to the another, $N_s, N_t \subset N$ are respectively the sets of start and end points of the function, and ι is a function associating every node $n \in N$ with the basic block $\iota(n)$.

Figure 3.1: Running Example: NULL Pointer Dereference Vulnerability and its Patch from OpenSSL 1.0.1l.

Notice that there might be more than one start point for a function, because it is hard to precisely identify the boundary of a function at the binary level [61].

Definition 3.3. Given a CFG $G = (N, E, N_s, N_t, \iota)$ of a function, a *partial trace* t in G is a finite sequence of nodes $\langle n_1, n_2, \dots, n_k \rangle$ for some $k \geq 1$, where $(n_i, n_{i+1}) \in E$, for every $i : 1 \leq i < k$.

3.2.2 Running Example

Figure 3.1 lists the partial X86 assembly code of one NULL pointer dereference vulnerability abstracted from the OpenSSL 1.0.1l that consists of the original version and patched version. In the original version, the program first calls the function `BN_new` by which the return value is stored in the register `eax`. Then, it assigns the return value to the register `edi` at the location 1. Later, the return value is directly used to dereference the memory `[edi+8]` at the location 2. This vulnerability is patched in OpenSSL 1.0.1m by checking whether the return value of the function call `BN_new` is NULL or not. This checking is implemented in the patched version by adding two instructions: `test eax, eax` and `jz error`. It first tests the return value at the location *a* by `test eax, eax` which performs a bitwise AND operation on the operand `eax`. The test operation sets the carry flag `cf` and overflow flag `of` to 0. The sign flag `sf` is set to the most significant bit of the result of the bitwise AND operation. The zero flag `zf` is set to 1 if the result

Figure 3.2: Framework Overview of SPAIN

of AND operation is 0, 0 otherwise. The parity flag **pf** is set to 1 if the number of ones in that byte is even, 0 otherwise. The value of the adjust flag **af** is undefined. After assigning the return value to the register **edi**, it checks whether the zero flag **zf** is 1 or not (i.e., the value of **eax** is Null or not) at the location *b*. If it is 1 (i.e., **eax** is NULL), the control flow will jump to the location **error** for error handling. Otherwise, the return value is used to dereference the memory [**edi+8**] at the location *c*.

In this running example, there are one basic block BB_1 and three basic blocks BB_1 , BB_2 and BB_3 respectively in the original version and patched version. The CFG of the original version is $G_o = (\{n_1\}, \emptyset, \{n_1\}, \{n_1\}, \iota)$, where $\iota(n_i) = BB_1$. The CFG of the patched version is $G_p = (\{n_1, n_2, n_3\}, \{(n_1, n_2), (n_1, n_3)\}, \{n_1\}, \{n_2, n_3\}, \iota)$, where $\iota(n_i) = BB_i$ for every $i : 1 \leq i \leq 3$. There are two partial traces $\langle n_1, n_2 \rangle$ and $\langle n_1, n_3 \rangle$ in G_p , which corresponds to two possible execution traces of the patched program. While there is only one partial trace $\langle n_1 \rangle$ in G_o .

3.2.3 Framework Overview

Figure 3.2 presents the overview of SPAIN, which consists of four components. Taking the original and patched versions of a binary program as inputs, SPAIN first locates the functions that have been changed from the original version to the patched version and detects the changed traces in each patched function to capture the function-level changes. Then SPAIN determines whether such changes are caused by security or non-security patches through semantic analysis and then summarizes the patterns of security patches and their corresponding vulnerabilities through taint analysis. As we run more binary programs, SPAIN

has the capability to continuously learn and accumulate the knowledge of security patches to embrace the evolving and emerging of various vulnerabilities.

Locating Patched Functions. Given the original and patched versions of a binary program, this component first uses the disassembler tool IDA Pro [62] and the binary comparison tool BinDiff [63] to obtain matched function pairs in the original and patched versions. Such pairs are called *candidate pairs* that may contain security or non-security patches. To reduce the size of pairs for further analysis and hence improve the scalability, this component removes the pairs from the candidate pairs, in which the functions have no changes or only compiler-introduced changes.

Identifying Patched Basic Blocks. For each function pair in the candidate pairs (i.e., the patched and the original functions), this component first leverages the pairwise basic block matching to identify the patched basic blocks within the patched function, identifies the relationships among these patched basic blocks in terms of patched partial traces (see Definition 3.3) that capture the locality of a patch, and finally determines the original partial traces in the original function that are relevant to each patched partial trace.

Identifying Security Patches. For a patched partial trace and its corresponding original partial traces, this component decides whether the changes are caused by a security or non-security patch. It is realized by a semantic analysis to compute the semantic difference between the patched partial trace and each of the original partial trace. We use small semantic difference as the indicator for security patches based on the heuristic that security patches are less likely to introduce new semantics than non-security patches (e.g., feature upgrades).

Example 3.1. *Recalling the running example in Figure 3.1, the difference between the original and patched versions is the sanity check, namely `test eax, eax` and `jz error` in the patched version. The semantic difference between the patched and original versions is very minimal (e.g. < 0.2 in our experiments) and hence, we can safely conclude that it is a security patch (see Section 3.3.3 for details).*

Summarizing Patch and Vulnerability Patterns. Once a security patch is identified, this component summarizes the patch pattern and the corresponding vulnerability pattern through a taint analysis from security-sensitive instructions in the patched function. The patterns capture the security-critical sources, sinks and sanity checks. One potential application of the summarized patterns is to search for similar patches or vulnerabilities in binary programs.

Example 3.2. *Consider the running example, the function `NB_new` is an external unknown function. The return value of `NB_new` is therefore regarded as the taint source/input. This taint source is directly used to perform the security-sensitive operation, memory dereferencing `[edi+8]`, which is regarded as the taint sink. There is no checking of the tainted source between the source and sink points. From this vulnerability, the vulnerability pattern summarized by SPAIN is shown in the 4th row (NULL Pointer Dereference) of Table 3.4. Intuitively, this pattern specifies that there is an untrusted function call whose return value is directly used to dereference memory without any sanity checking on them.*

While in the patched version of the running example, the return value from the external unknown function `NB_new` is checked for `NULL` value. If it is `NULL`, the program jumps to an error handler. Otherwise, the return value is used to perform the security-sensitive operation, that is memory dereferencing. The patch pattern learned by SPAIN is shown in the 4th row (NULL Pointer Dereference) of Table 3.4. Intuitively, this patch pattern expresses that before using the return value of an untrusted function call to dereference memory, there must be a sanity checking of the return value.

3.2.4 Assumptions

SPAIN has a few underlying assumptions, which may limit its application and threat its validity. First, we focus on patches in which only one function is modified for one patch, but do not support patches where multiple functions are changed for one patch. Second, we assume that the function matching results generated

by IDA Pro [62] and BinDiff [63] are correct, although no tools can reach 100% accuracy [64]. Third, we cannot identify every type of patches since some patches, especially for logical vulnerability patches, may behave like the normal code. Currently, we cover the patches for most common vulnerabilities such as buffer overflows, integer overflows, and double-free/use-after-free. Fourth, we focus on the X86 32bit binary format. We will discuss these assumptions in Section 3.5.

3.3 Methodology

In this section, we elaborate each component depicted in Figure 3.2.

3.3.1 Locating Patched Functions

SPAIN starts with locating the matched pairs of functions $\mathcal{F} = \{\langle f_p, f_o \rangle \mid f \text{ is a function in a binary program}\}$, where f_o (i.e., the original function) is changed to f_p (i.e., the patched function) during the patching process. $\mathcal{F}$ is a set of candidate pairs that may contain security or non-security patches.

3.3.1.1 Function Matching

In the first step, we leverage the disassembler tool IDA Pro [62] and the binary comparison tool BinDiff [63] to match functions in the original and patched versions of a binary program. In particular, we use IDA Pro to extract the assembly instructions and construct the CFG for each function. Then, for open source binaries (i.e., with the symbol table), we directly use function names to perform the matching. While for closed source binaries (i.e., without the symbol table), we leverage BinDiff's function matching functionality. Finally, we get the initial candidate pairs, where each function is represented by its CFG. Note that this step is not one of our contributions, and we briefly introduce it to make our methodology complete.

Figure 3.3: An Example of Compiler-Introduced Changes

3.3.1.2 Function Filtering

Patches usually change a small part of the whole binary program and many functions remain the same. Therefore, in the second step, we remove the function pairs with no changes from the candidate pairs to reduce the size of pairs for further analysis and thus improve the scalability. To this end, we apply the 3D-CFG-based hashing technique [65] to compute the hash value for f_p and f_o , and then remove $\langle f_p, f_o \rangle$ from the candidate pairs $\mathcal{F}$ if f_p and f_o have the same hash value.

Further, some changes are introduced by compilers due to the compilation context or optimization level [66]. One common case is that the compiler may split one basic block into two basic blocks, as illustrated in Figure 3.3. The basic block in Figure 3.3(a) becomes two basic blocks connected by a `jmp` instruction in Figure 3.3(b) in the second run of a compiler. However, these two versions are semantically equivalent to each other. Therefore, for any basic block, which has only one successor and its successor has only one predecessor, we merge it with its successor and remove the additional `jmp` instruction.

Another common case is that the compiler might change the operands of the instruction in different runs. For example, the memory addresses are associated with the function position in the binary program and they will change if the function position is changed, which always happens in different runs of a compiler. To account for such changes introduced by compilers, we normalize each basic block, as shown in Algorithm 1. It takes a basic block as an input and returns the normalized basic block. It iteratively normalizes each assembly instruction in the basic block (Line 2). An assembly instruction consists of a mnemonic and

Algorithm 1: Normalize A Basic Block

```

Input  : basic block  $b$ 
Output: normalized basic block  $b'$ 
1  $b' := \langle \rangle$  // sequence of normalized instructions
2 foreach instruction  $i$  in  $b$  do
3    $m := \text{GetMnemonic}(i)$ 
4   if  $m \neq \text{'nop'}$  then
5      $op' := \langle \rangle$  // sequence of operand type
6     foreach operand  $o$  in  $\text{operands}(i)$  do
7        $t := \text{GetOperandType}(o)$ 
7       //  $t \in \{\text{mem}, \text{reg}, \text{imm}\}$ 
8        $op' := op' \oplus t$ 
9     end
10     $i' := \langle m, op' \rangle$  // normalized instructions
11     $b' := b' \oplus i'$ 
12  end
13 end
14 return  $b'$ 

```

up to 3 operands, where a mnemonic represents the specific operation that an instruction performs while an operand is a variable-length sequence of elements (cf. IA-32 [67]). Thus, it gets the mnemonic (Line 3) and normalizes the operands of each instruction according to their operand types (Lines 6–9). An operand can be a register, an immediate value, or a memory, which is respectively normalized to its type `reg`, `imm`, or `mem`. For example, `mov eax, 0x40` is normalized to `mov reg, imm`.

After the merging and normalization of basic blocks for each functions in $\mathcal{F}$, we use the same hashing technique to further remove the pairs that only have some compiler-introduced changes. Our filtering step is designed to be conservative such that security changes are kept.

3.3.2 Identifying Patched Basic Blocks

Both security and non-security patches can lead to the modifications of various basic blocks and, in the worst case, such modified basic blocks can lie scattered all over a function. Thus, SPAIN proceeds to investigate the candidate pairs $\mathcal{F}$ to identify the basic blocks that are modified (i.e., patched basic blocks) in the patch,

Algorithm 2: Identify Partial Traces

Input : patched function f_p , original function f_o
Output: set of matched partial trace pairs $\mathcal{T}$

```

1  $\mathcal{T} := \emptyset$ 
2  $\mathcal{B}_p := \emptyset$  // set of patched blocks
3 foreach basic block  $b_p$  in  $f_p$  do
4   foreach basic block  $b_o$  in  $f_o$  do
5     if  $b_p == b_o$  then
6        $FoundMatch = True$ 
7       break
8     end
9   end
10  if not  $FoundMatch$  then
11     $\mathcal{B}_p := \mathcal{B}_p \cup \{b_p\}$ 
12  end
13 end
    // link patched basic-blocks based on pred/succ information
14  $\mathcal{T}_p := \text{LinearConnectedComponents}(\mathcal{B}_p)$ 
15 foreach patched partial trace  $t_p$  in  $\mathcal{T}_p$  do
16    $neighbors := \text{GetFirstDegreeNeighbors}(t_p, f_p)$ 
17    $\mathcal{B}_o := \text{GetRelevantOriginalBlocks}(neighbors, f_o)$ 
18    $\mathcal{T}_o := \text{LinearConnectedComponents}(\mathcal{B}_o)$ 
19    $\mathcal{T} := \mathcal{T} \cup \{\langle t_p, \mathcal{T}_o \rangle\}$ 
20 end
21 return  $\mathcal{T}$ 

```

identify the relationships among these patched basic blocks in terms of partial traces (see Definition 3.3), and identify their relations to the original function. Algorithm 2 gives this procedure, which computes the matched trace pairs $\mathcal{T} = \{\langle t_p, \{t_o^1, \dots, t_o^n\} \rangle \mid t \text{ is a partial trace in a function} \}$ for each function pair $\langle f_p, f_o \rangle$ in $\mathcal{F}$, where t_p in f_p might be relevant to $\{t_o^1, \dots, t_o^n\}$ in f_o . Here, t_p and t_o represent partial traces in patched and original version of the binary, respectively.

In detail, we leverage the pairwise basic block matching to identify the patched basic blocks within the patched function (Lines 2–13). In particular, for each basic block b_p in the patched function f_p , we search for an equivalent basic block b_o in the original function f_o . If there is no such a matching basic block b_o in f_o , b_p is identified as a patched basic block.

Once the patched basic blocks are identified, we proceed to determine the relationships among these basic blocks (Line 14), i.e., to connect the patched basic blocks

to infer the effect of a patch. To this end, for each patched basic block, we leverage the predecessor/successor information to connect related patched blocks; i.e., b_p^1 and b_p^2 are connected when there is a predecessor/successor relationship between them. In such a manner, patched partial traces (i.e., $\{t_p\}$) are constructed.

To get a clear understanding of how a patched partial trace t_p is related to the original function, we extract the unmodified neighbouring basic-blocks of the patched partial trace (Line 16). Specifically, for each basic block in the patched partial trace, we identify the first-degree neighboring blocks that are unmodified from the original function. These unmodified neighboring basic blocks capture the locality of the patch and help to locate the corresponding basic blocks in the original function (Line 17). In this fashion, for each patched partial trace, we identify the corresponding basic blocks of interest in the original function, and construct the original partial traces $\{t_o^1, \dots, t_o^n\}$ in the same way to construct patched partial traces (Line 18).

Example 3.3. *Consider the original and patched functions in Figure 3.4, where each letter represents a basic block. We can see that in the patched function, the basic block b in the original function is modified to b' and a new basic block g' is added. From the patched function, only one patched partial trace can be extracted, i.e., $\langle b', g' \rangle$, where its unmodified first-degree neighbors are $\{a, c, d\}$. By looking at the first-degree neighbors of the patched partial trace, we can infer that only the basic block b is the corresponding basic block of interest in the original function, and only one original partial trace can be constructed, i.e., $\langle b \rangle$. Hence, in this case, the patched partial trace $\langle b', g' \rangle$ is only related to one original partial trace $\langle b \rangle$.*

3.3.3 Identifying Security Patches

Once the patched partial traces and their related partial traces in the original function are identified, we proceed to determine for each pair $\langle t_p, \{t_o^1, \dots, t_o^n\} \rangle$ in $\mathcal{T}$ whether the changes are caused by a security patch or non-security patch. To this

Figure 3.4: An Example of the Original and Patched Functions

end, we perform a semantic analysis on both the patched partial trace t_p and the set of original partial traces $\{t_o^1, \dots, t_o^n\}$.

The idea underlying our semantic analysis is that “a security patch is less likely to change the semantics of the underlying function, while a non-security patch is more likely to introduce new semantics”. Therefore, we compare the *semantic summaries* generated for the patched partial trace and each of the original partial traces by Equation 3.1.

$$\delta_d = \mathcal{S}_p - \mathcal{S}_o \quad (3.1)$$

δ_d is the semantic difference between the semantic summary $\mathcal{S}_p$ of the patched partial trace t_p and the semantic summary $\mathcal{S}_o$ of the original partial trace t_o in $\{t_o^1, \dots, t_o^n\}$. If there exists one original partial trace such that the semantic difference is small, we mark the patch as a security patch. Otherwise, we mark the patch as a non-security patch.

In detail, we leverage the technique in our previous work [68] to generate the semantic summary from a partial code segment (i.e., a partial trace). Here semantics are expressed as the effects of executing the partial code segment on the machine state. The machine state s is characterized by a 3-tuple $\langle Mem, Reg, Flag \rangle$, denoting the memory Mem , the general-purpose registers Reg , and the condition-code flags $Flag$. The machine state before and after executing the partial code segment is referred to as *pre-state* and *post-state*, respectively. For example, one possible pre-state before executing the code segment in Figure 3.5(a) is given in Figure 3.5(b), where all **registers**, **flags**, and **memory** are assigned by the value

0; and in the corresponding post-state, the registers `eax` and `ebx` hold the values 0x04 and -0x04, respectively, while the sign flag `sf` holds the value 1 due to the negative result in `ebx`.

Then, the semantic summary is the difference between the pre-state and post-state, as shown in Equation 3.2.

$$\mathcal{S} = s_{post} - s_{pre} \quad (3.2)$$

For example, the semantic summary of the code segment shown in Figure 3.5(a) is $eax' = 0x04$ and $ebx' = eax - 0x04$, where primed variables denote final values and non-primed variables denote initial values.

In our semantic analysis, for both patched and original partial traces, we first generate various configurations of pre-state and run the partial traces and measure the corresponding post-state values. Then, we compute the semantic summary for patched and original partial traces, and compare them following the techniques in [68, 69]. Finally, if the semantic difference is below a pre-defined threshold value (i.e., $< \Delta_d$). We determine that the patch is a security patch. Otherwise, it is a non-security patch. In this study, based on our manual analysis of 50 security and non-security patches, we empirically fix Δ_d to be 0.20.

Example 3.4. *Let us consider the running example in Figure 3.1. There are in total 17 machine artifacts involved in the semantic summary computation that are 8 register, 8 status flags and one memory location `[edi+8]`, among which only three status flags (`sf`, `zf` and `cf`) are influenced by the newly added instruction `test eax, eax` in the patched partial trace. Given a fixed pre-state in which all values of artifacts are set to 0 but `ebx` to 4, as shown in Figure 3.5(b), the post-state can be computed easily. In the post-state, only the zero flag `zf` is set to 1, while all the other 16 artifacts keep same as the original trace. From the pre-state and post-state, we get that the semantic difference between the patched and original partial trace is equal to 0.058 ($< \Delta_d$), which implies that the patch in Figure 3.1 is a security patch.*

Figure 3.5: An Example of Pre-State and Post-State

3.3.4 Summarizing Patch and Vulnerability Patterns

Once the security patches are identified, SPAIN proceeds to summarize patch patterns and the corresponding vulnerability patterns from the original and patched partial traces. To this end, we introduce a light-weight program analysis technique. Specifically, given a patched partial trace t_p and the relevant original partial traces $\{t_o^1, \dots, t_o^n\}$, SPAIN identifies the newly-added and security-sensitive instructions in t_p . In general, newly-added control transfer instructions, especially the ones that depend on comparison instructions such as **cmp** and **test**, are of the interest for security analysts, which are more likely to reflect the newly-introduced sanity checks in the patch.

In the next step, we pass the source and destination operands of those interesting instructions to the taint engine [70] to track their sources and sinks that are key indicators of vulnerabilities. For example, in the instruction **cmp eax, ebx**, register **ebx** is the source operand and register **eax** is the destination operand. In particular, non-immediate source and destination operands are passed to the taint engine to track their origins (or *sources*) using backward taint analysis, while the non-immediate destination operand is passed to the taint engine to track its destinations (or *sinks*) using forward taint analysis. It is important to note that taint analysis is performed within the patched function, i.e., intra-procedural taint analysis.

Figure 3.6: Abstract Vulnerability and Security Patch Pattern

Finally, the tracked sources and sinks are combined to summarize the vulnerability and security patch patterns. In Figure 3.6, we show the abstract vulnerability and patch patterns, where *source*, *sink* and *sanity check* are defined as follows.

Definition 3.4 (Sources). *Taint sources* are the user/external inputs (i.e., tainted inputs) that can reach the patched function and are used by those interesting instructions.

For example, external function parameters or return values of security-sensitive system APIs (e.g., `scanf`) are considered as taint sources.

Definition 3.5 (Sinks). *Taint sinks* are the security-critical operations that involve the tainted inputs.

For example, memory dereference operations (e.g., `mov eax, [taint-source]`) or arithmetic subtraction operations (e.g., `mov eax, [taint-source]; sub ebx, eax`) that involves taint sources are considered as taint sinks.

Definition 3.6 (Sanity Checks). *Sanity checks* are operations performed on the tainted inputs before they are involved in security-critical operations.

3.3.5 Applications of Patterns

The patch patterns can be used in many applications. One main application is to search for the similar patches and corresponding vulnerabilities in the binaries. Besides, SPAIN is orthogonal to several patch analysis tools, and hence can

provide patch patterns as input for them. For example, Prophet [53] can learn a probabilistic model from the correct code to automatically generate patches. SPAIN can provide possible locations where a patch is needed. TEDEM [71] can identify binary code regions that are similar to code regions containing vulnerabilities. SPAIN can provide such vulnerable regions through pattern matching so that TEDEM may have more candidates to search for. Honey patch [60] was proposed as a trap to monitor attack information and misinform the attacker through redirecting the attack to an unpatched decoy. SPAIN can help to identify the patches that can be converted into honey patches, making the whole process fully automatic. A survey on repeated patches [72] has been conducted to show the general trend of bug fixes. SPAIN can enable the trend analysis on a large number of programs to gain a complete understanding of how programmers fix bugs.

3.4 Evaluation

In this section, we conduct an experimental study on several real-world projects to answer the following research questions.

- **RQ1:** What is the accuracy and scalability of SPAIN to identify security patches?
- **RQ2:** What are the security patch patterns and their corresponding vulnerability patterns summarized by SPAIN?
- **RQ3:** What are the potential application scenarios of the summarized patterns of SPAIN?

The experiments were conducted on an HP z420 workstation with 32GB RAM and Intel Xeon CPU E5-1620 v2 3.70GHz. All the experimental data is available at our website [73]. We used the following real-world software in our evaluation:

- **OpenSSL** is an open source software with around 446,747 number of locations (LOC) and developed since 1998.

- **Linux Kernel** is an open source software with around 18,963,973 LOC and developed since 2002.
- **Adobe PDF Reader** is a closed source software. We use two of its libraries, `3difr.x3d` and `AXSLE.dll`, which have around 1,293 and 4,874 functions respectively.

3.4.1 Accuracy and Scalability (RQ1)

3.4.1.1 Accuracy

To evaluate the accuracy of SPAIN on identifying security patches, we manually identified all the security and non-security patches of all the 20 versions of OpenSSL 1.0.1 by analyzing its commits on GitHub. They were used as the ground truth to evaluate the true positive and false positive of SPAIN. For each security patch, we also manually analyzed the type of the patched vulnerability.

Table 3.1 reports the detailed results of our accuracy evaluation on OpenSSL. The first column lists the version numbers of two consecutive versions of OpenSSL, which are respectively served as the original and patched binaries. The second column reports the number of CVEs that are documented. The third and fifth columns respectively show the number of security and non-security patches we manually identified. The fourth column reports the true security patches SPAIN successfully identified, and the sixth column gives the false security patches SPAIN *incorrectly* identified. Note that SPAIN only reports security patches, here the sixth column gives the false positive cases generated by SPAIN. The last three columns compute the true positive, false positive and execution time of SPAIN.

From Table 3.1 it can be seen that among the 323 security patches, SPAIN successfully identified 229 of them, while it incorrectly identified 47 of the 217 non-security patches as security patches. It achieved the true positive rate of 71% and the false positive rate of 22%, which indicated that SPAIN can identify security patches with high accuracy. Besides, compared with the number of identified

Table 3.1: Accuracy and Performance on OpenSSL

Ver.	CVE	Sec. Pat.		Non-Sec. Pat.		T.P.	F.P.	T. (s)
		G.T.	Iden.	G.T.	Iden.			
0-a	1	7	4	11	0	0.57	0	11
a-b	0	0	0	5	0	–	0	13
b-c	1	3	2	0	0	0.67	–	7
c-d	3	19	8	27	0	0.42	0	60
d-e	0	0	0	6	1	–	0.17	8
e-f	3	12	9	17	10	0.75	0.59	14
f-g	2	10	8	6	1	0.8	0.17	11
g-h	8	29	18	4	1	0.62	0.25	20
h-i	9	37	24	23	8	0.65	0.35	54
i-j	4	23	11	5	2	0.48	0.4	35
j-k	7	25	19	15	5	0.76	0.33	39
k-l	0	0	0	1	0	–	0	7
l-m	8	34	25	7	3	0.74	0.43	95
m-n	5	60	49	38	5	0.82	0.13	76
n-o	1	0	0	6	0	0	0	9
o-p	2	2	1	1	0	0.5	0	9
p-q	2	39	31	40	11	0.79	0.28	48
q-r	1	10	9	3	0	0.9	0	18
r-s	6	13	11	2	0	0.85	0	36
Sum.	63	323	229	217	47	0.71	0.22	–

security patches, only a small number of CVEs are documented, which demonstrates that SPAIN can discover undocumented patches. Moreover, SPAIN took around 32 seconds on average to analyze the binaries. In addition, Table 3.2 shows the accuracy of SPAIN with respect to nine vulnerability types. Note that 109 of the security patches patched some tricky vulnerabilities that do not belong to these nine common types. The results indicate that SPAIN can identify security patches for different types of vulnerabilities with high accuracy.

Table 3.2: Accuracy on OpenSSL w.r.t Vulnerability Types

Vulnerability Type	G.T.	SPAIN	Accuracy
Buffer Overflow	54	37	0.69
NULL Pointer Dereference	65	61	0.94
Memory Leak	51	30	0.59
Double-Free	14	7	0.5
Integer Overflow	8	7	0.88
Initialization	16	11	0.69
Off-by-One	3	2	0.67
Side Channels	2	1	0.5
Use-After-Free	1	1	1
Others	109	72	0.66
Sum.	323	229	0.71

By closely looking into the security patches SPAIN failed to identify, we find two main causes for the false negatives. First, a patch is so simple that our function filtering step may fail to detect the changes. For example, one unidentified security patch in OpenSSL simply increased the buffer size in a function by a constant value. As a result, the patched function is the same to the original one, except for that particular constant. After the basic block normalization, they become the same and will not be further analyzed. Second, a patch is so complicated that our semantic analysis may identify it as a non-security patch due to the large amount of newly-introduced semantics. For example, developers may rewrite part of a function to fix a vulnerability; or some different patches happen to patch the same function. In such cases, our semantic analysis may detect significant semantic difference between the patched function and the original one, failing to identify the security patch.

Similarly, we investigated the causes of the false positives. One main reason is that some non-security patches only slightly modify the program, especially for fixing some performance bugs [74] or adding the consideration for some missed corner cases. For example, a patch added a simple conditional statement that

would be executed only when certain criteria have been met. It follows the similar pattern of security patches since most security patches can be seen as a conditional functionality, which redirects the execution to safe places if some variables have unexpected values. Therefore, such kinds of non-security patches are difficult to be distinguished. We argue that false positives only have a small impact on our analysis as the number of false positive is small, and a simple manual validation can identify them.

Summary. Based on these observations, we can positively answer RQ1 that SPAIN can identify security patches for different types of vulnerabilities with a high true positive rate as well as acceptable false positive rate. Further, SPAIN can discover security patches that are not documented.

3.4.1.2 Scalability

To evaluate the scalability of SPAIN to analyze large binaries, we ran SPAIN on both open source Linux Kernel and closed source Adobe PDF Reader. In particular, we used versions 3.16.2 and 3.16.3 of Linux Kernel, and compiled them using the `-O2` optimization level (i.e., the most common commercial setting) with all the functions included. For Adobe PDF Reader, we analyzed the library `3dofr.x3d` of versions 11.0.08, 11.0.09, 11.0.13, 11.0.14, 11.0.15 and 11.0.16 as well as the library `AXSLE.dll` of versions 11.0.15 and 11.0.17.

Table 5.8 presents the performance of SPAIN on Linux kernel and Adobe PDF Reader. The first column gives the versions of the original and patched binaries, and the second column reports the average number of functions in them. The third and fourth columns list the identified security patches by SPAIN and the corresponding time overhead. We can see that SPAIN analyzed the whole Linux Kernel in 807 seconds, and analyzed the two Adobe libraries in 23 seconds. Note that, because we do not have the ground truth for security patches, we did not show the accuracy of SPAIN on them but reported the identified security patches.

Table 3.3: Performance on Linux and Adobe PDF Reader

Version	Total Func.	Sec. Patched	T. (s)
Linux: 3.16.2 – 3.16.3	249341	1221	807
difr.x3d: 11.0.08 – 11.0.09	1293	12	21
difr.x3d: 11.0.13 – 11.0.14	1293	13	23
difr.x3d: 11.0.15 – 11.0.16	1293	11	20
AXSLE.dll: 11.0.15 – 11.0.17	4875	27	84

Summary. Based on the results from Table 5.8, we can positively answer RQ1 that SPAIN scales well to large binaries.

3.4.2 Patch and Vulnerability Patterns (RQ2)

Table 3.4 presents the vulnerability and patch patterns summarized for the key vulnerability types patched in the 20 versions of OpenSSL. Among them, one of most common ones is double-free vulnerability, i.e., an error that occurs when `free()` is called more than once with the same memory address as an argument. It is summarized in the first row, where the memory address is obtained from an untrusted function as a return value, and it is freed more than once, which leads to the vulnerability. To patch this, one needs to sanity check for validity of the memory address and remove all the occurrences where it is freed for more than once.

The integer overflow/underflow is summarized in the second row, where it occurs when an arithmetic calculation produces a result that is greater (or smaller) in magnitude than that a given register or storage location can store or represent. In general, arithmetic operations are vulnerable to integer overflow or underflow when they take the inputs from untrusted sources and perform some security-sensitive operation such as memory dereferencing and memory indexing on the calculated results. To patch such vulnerable cases, one needs to perform a sanity check on the untrusted inputs before allowing for any security-sensitive arithmetic operation.

The third row summarizes the use-after-free vulnerability pattern, where the obtained pointer (from an untrusted source) to the memory object is freed without checking for liveness property of the pointer. For patching such vulnerabilities, one needs to check whether the object is in use, if so, the pointer to it should not be freed. The fourth row summarizes NULL pointer dereference vulnerability, the most frequently occurred vulnerability in OpenSSL. As discussed in Section 5.2, it is very common to obtain a pointer from an untrusted source; hence, before involving it in any security-sensitive operation, such as memory dereferencing, it needs to be checked whether the pointer is NULL or valid. Finally, the fifth row summarizes the other most commonly observed vulnerability, buffer overflow or underflow, where the patch suggests that any pointer to a memory object should be properly bounds-checked before involving it in any security-sensitive operation.

Apart from the vulnerability types summarized in Table 3.4, we observed other class of vulnerabilities/patches that cannot be generalized for pattern matching. Such vulnerability types include, side-channel information leakage, memory leakage and uninitialized variables whose patterns are particular to the OpenSSL binaries. However, summarizing these pattern will enable us to identify clone or copy-paste type vulnerabilities that are very commonly observed in the wild [69, 75]. Due to the space limitation, we provide the vulnerability and patch patterns for such class of vulnerabilities and the patterns for Linux and Adobe in our website [73].

Table 3.4: Patch Patterns vs. Vulnerability Patterns, where TNT denotes taint, IN denotes input, and SSTV denotes sensitive.

Vul. type	Concrete Vulnerability	Vulnerability Pattern	Concrete Patch	Patch Pattern
Double Free	<pre> call BN_to_ASN1_INTEGER test eax, eax ... mov [esp + 4Ch + var_4C], esi call ASN1_STRING_clear_free ... mov [esp + 4Ch + var_4C], esi call ASN1_STRING_clear_free </pre>	<pre> call (untrusted_func) <sanity check> : (return_value) ... mov (func_param), (return_value) call (free) ... mov (func_param), (return_value) call (free) </pre>	<pre> call BN_to_ASN1_INTEGER test eax, eax mov esi, eax ... mov [esp + 4Ch + var_4C], esi call ASN1_STRING_clear_free ... </pre>	<pre> call (untrusted_func) <sanity check> : (return_value) ... mov (func_param), (return_value) call (free) ... </pre>
Integer Underflow Overflow	<pre> mov eax, [esp + 4Ch + arg_4] ... mov ecx, eax sub ecx, [esp + 4Ch + var_2C] add [esp + 4Ch + var_24], ecx ... mov esi, [esp + 4Ch + _24] mov ecx, [esp + 4Ch + arg_8] mov [ecx], esi </pre>	<pre> mov (TNT_IN), (untrusted_src) ... <arith.op> (TNT_result), (TNT_IN) ... mov (sec.SSTV.sink), (TNT_result) </pre>	<pre> mov eax, [esp + 4Ch + arg_4] ... mov edi, [esp + 4Ch + var_2C] cmp eax, edi jl error mov ecx, eax sub ecx, [esp + 4Ch + var_2C] add [esp + 4Ch + var_24], ecx ... mov esi, [esp + 4Ch + _24] mov ecx, [esp + 4Ch + arg_8] mov [ecx], esi </pre>	<pre> mov (TNT_IN), (untrusted_src) <sanity check> : (TNT_IN) ... <arith.op> (TNT_result), (TNT_IN) ... mov (sec.SSTV.sink), (TNT_result) </pre>
Use After Free	<pre> mov ebp, [esp + 0ECh + arg_0] ... mov [esp + 0ECh + dest], ebp call ssl3_release_read </pre>	<pre> mov (TNT_IN), (untrusted_src) ... mov (func_param), (TNT_pointer) call (free) </pre>	<pre> mov ebp, [esp + 0ECh + arg_0] ... mov eax, [ebp + 58h] mov ecx, [ebp + 0F8h] test eax, ecx jnz (do not release) ... mov [esp + 0ECh + dest], ebp call ssl3_release_read </pre>	<pre> mov (TNT_IN), (untrusted_src) <sanity check> : (TNT_pointer) ... mov (func_param), (TNT_pointer) call (free) </pre>
NULL Pointer Dereference	<pre> call BN_new mov edi, eax ... cmp eax, [edi + 8] </pre>	<pre> call (untrusted_func) ... <mem.deref> : (return_value) </pre>	<pre> call BN_new test eax, eax mov edi, eax jz error ... cmp eax, [edi + 8] </pre>	<pre> call (untrusted_func) <sanity check> : (return_value) ... <mem.deref> : (return_value) </pre>
Buffer Overflow	<pre> mov ebx, [esp + 3Ch + arg_8] ... mov [esp + 3Ch + n], ebx call eax </pre>	<pre> mov (TNT_IN), (untrusted_src) mov (func_param), (TNT_IN) ... call (untrusted_func) </pre>	<pre> mov ebx, [esp + 3Ch + arg_8] cmp ebx, [esp + 3Ch + arg_4] jl error ... mov [esp + 3Ch + n], ebx call eax </pre>	<pre> mov (TNT_IN), (untrusted_src) <sanity check> : (TNT_IN) ... mov (func_param), (TNT_IN) call (untrusted_func) </pre>

3.4.3 Application of Patterns (RQ3)

Using SPAIN and the summarized patterns, we used pattern matching techniques [68] to discover three zero-day vulnerabilities (CVE-2016-0933, CVE-2016-1037 and CVE-2016-4198) in the two libraries in Adobe PDF Reader.

We start the experiment with an Adobe Reader vulnerability, CVE-2014-0565, which has already been patched in 2014. It is a vulnerability that reads the memory out of bound, due to the insufficient check on the string length of *Line Set* block. Our tool successfully identifies the patch by diffing the two versions of Adobe Reader, 11.0.8 and 11.0.9, i.e., finding the place where two additional checks have been added to the function. After the checks, the program returns to the normal execution. Therefore, our tool identifies security patches with high confidence. It is a very typical patch pattern, which is the buffer overflow pattern in the fifth row of Table 3.4.

Then, we use the corresponding vulnerability pattern to search for similar vulnerabilities in different versions of Adobe Reader. Specifically, we find CVE-2016-0933 and CVE-2016-1037, having the same vulnerability pattern, which, later, have been confirmed by manual analysis. CVE-2016-0933 is a vulnerability in Adobe Reader 3diffr.x3d before version 11.0.14. CVE-2016-1037 also resides in 3diffr.x3d, while it is in version 11.0.16. We also find CVE-2016-4198, which shares the similar pattern with the previous vulnerability, although it is indeed an integer overflow vulnerability in Adobe XSLT library. Once triggered, it will cause an out of bound write to the memory and remote code execution.

After Adobe Reader had patched the aforementioned vulnerabilities, we used SPAIN to diff the patched version with the original vulnerable version. We successfully located the three vulnerability patches using the tool, as shown in Table 5.8, which demonstrates that our tool has the capability to capture patches in closed source binaries. A more detailed explanation of the patterns can be found at our website [73].

3.5 Discussion

Our framework has the following limitations. First, our framework tries to look for patches and vulnerabilities with patterns. However, some real-world vulnerabilities are actually the corner cases, which have unique signatures and patches may fix bugs in unusual ways. SPAIN may not be able to achieve high accuracy when dealing with these special cases, in which almost all the patches are usually complex.

Second, vulnerabilities may be patched in some places other than the places where they are triggered. Therefore, trying to hunt them from patches may not be straightforward. Our current tool can only look for vulnerabilities that are patched within the one function where the patch has been found. In future, we plan to adopt some other techniques such as function summarization, more advanced slicing and symbolic execution to handle the vulnerabilities that are patched across functions.

Third, building a general solution for all kinds of binary architectures requires lifting the instruction into intermediate representation. We did not do that because we want to obtain the exact patterns of the patches and vulnerabilities. Binary lifting may result in loss of a certain amount of information or inaccuracy [76], which may be critical for our pattern summarization. Therefore, throughout this work, all the binaries used are in X86 32bit format. It is worth mentioning that our approach is general and can be adapted to other binary formats.

Fourth, the framework shares the common drawback of static vulnerability searching approaches. Although we can identify the location of the vulnerability, we cannot reproduce it in real world if the program is too complicated. The user input may travel through many functions and be transformed several times until it reaches the location that triggers the vulnerability. Hence, we may not have a concrete proof-of-concept (POC) to validate the vulnerability whether it has a real impact on the program's security. However, unlike other searching methods,

since we are searching the vulnerability based on its patch, we have a relatively high confidence to say that it is a severe bug, which is needed to be patched.

3.6 Related Work

Our work attempts to understand patches and the corresponding vulnerabilities, and summarize their patterns for discovering similar patches or vulnerabilities. Hence, we discuss the related work in the areas of patch analysis and vulnerability modeling.

3.6.1 Patch Analysis and Diffing

PVDF [77] computes the semantic of patches for privilege elevation vulnerabilities. The patch semantic is then used to guide fuzzy testing to discover new vulnerabilities in binary programs. It takes the vulnerable binary and the corresponding patch as the inputs, and leverages forward and backward taint analysis to extract the semantics. This work is similar to ours; but it assumes the availability of patches, and only focuses on one particular vulnerability type. Differently, SPAIN attempts to summarize patterns for different vulnerability types, and only requires the binary programs but not the patches.

BISSAM [59] uses binary patch diffing information to develop signatures for the security update. Then it executes the binary with malicious inputs under dynamic monitoring to obtain the execution paths. It identifies the vulnerability by determining whether the path includes one of the signatures. Its effectiveness heavily relies on the malicious inputs and assumes the availability of security patches. Differently, we can automatically identify the security patches in a static way.

BinHunt [58] can automatically find the semantic differences in binary programs. It lifts the binary instructions to Intermediate Representation (IR), and constructs CFGs at the IR level. Then it uses graph isomorphism to find the differences and performs symbolic execution on them to obtain the semantics for theorem proving.

Unlike it, SPAIN is much more scalable, since we apply function-level filtering, which enables us to search for patches in real-world programs.

Apart from these binary-level patch analysis, there are many source-level approaches. Tian et al. [56] attempt to identify Linux patches based on the commit messages and the source code diffing analysis. They extract features and leverage machine learning techniques to predict whether a commit is a bug fix or not. Soto et al. [78] investigate Java projects to get a deeper understanding of each patch's signature to guide automatic program repairing. BugTrace [57] aims at building the link between the bug and the fix through patch analysis. Kim and Notkin [79] build a diffing tool to infer the structural differences between codes, which helps programmers to discover bugs by comparing two versions of a program. These works all use patch analysis to gain the understanding at the source code level, which may not work if the source code is not available. Instead, SPAIN directly works at the binary level.

3.6.2 Vulnerability Modeling and Searching

INDIO [80] leverages symbolic execution to analyze binaries and detects integer overflow vulnerabilities. It uses some heuristic patterns to find the potential vulnerability candidates. Then it ranks the vulnerable possibility for the candidates. Finally, it selectively executes symbolic execution to remove the false positives further. It discovered known CVEs as well as unknown integer overflow vulnerabilities in real world Window binaries. Similarly, IntScope [81] employs symbolic execution to detect integer overflow vulnerabilities in X86 binaries, and Firmalice [82] uses symbolic execution to detect authentication bypass vulnerabilities in firmware.

Brumley et al. [83] showed that automatic patch-based exploit generation (APEG) was possible. They combine dynamic symbolic execution and static control flow graph analysis to summarize the constraint formulae of the vulnerabilities, which were successfully used to generate exploits for 5 real-world vulnerabilities.

GUEB [84] searches for use-after-free vulnerability patterns in the binary programs. It builds an abstract memory model for the binary functions. Then it uses value set analysis to reason each variable in the assignment and free instructions. If a variable is used after the free instruction, GUEB reports it as a vulnerability. It found one real-world use-after-free vulnerability in ProFTPD program. LoongChecker [85] also uses value set analysis to statically detect potential vulnerabilities in binaries.

The aforementioned works leverage program analysis methods to extract semantics of the binary program, and compare them with certain models to discover potential vulnerabilities. They require expertise to build such models specifically for a specific type of vulnerabilities. Differently, our work aims at automatically summarizing the patterns for different types of vulnerabilities and using them to search for vulnerabilities.

Apart from these binary-level vulnerability modeling and searching approaches, there are many works targeting vulnerability modeling at the source code level. Yamaguchi et al. [86] use a taint-style pattern to search vulnerabilities and filter out irrelevant code. It can reduce the code base by 94.9% on average to improve the code audit efficiency. Wagner et al. [87] reduce the searching of the buffer overflow vulnerabilities to an integer range analysis problem, which discovered vulnerabilities in the Sendmail software. Averinos et al. [88] proposed an automatic exploit generation tool and analyzed 20 open source programs. They use preconditioned symbolic execution to generate control flow hijack attacks and discovered 2 unknown vulnerabilities. However, those approaches work on the source code level; but they fail to analyze binary-level patches. Moreover, more attention has been paid to address the gap between the source code and the compiled binaries [89]. The compiler will introduce bugs even if the source code is correct. Therefore, we choose to work on binary level to be closer to the machine so that, ideally, the framework can capture all possible vulnerabilities.

3.7 Conclusion

In this chapter, we proposed a patch analysis framework SPAIN to automatically learn the security patch patterns and vulnerability patterns, and identify them from the program binary executables. It has built the bridge from the binary diffing to the automatic patch understanding. The experiments have shown that SPAIN can correctly locate more than half of the vulnerability patches in the binaries and find n-day vulnerabilities in major commercial software. SPAIN can be useful in vulnerability and patch understanding, similar bug hunting, binary code auditing, and, eventually, the program security enhancement. In the future, we plan to extend the framework to generate more detail and precise report of each patch and vulnerability through program slicing and symbolic execution, as well as to make it capable of analyzing binaries with other instruction sets, like ARM. Also, we would like to set up binary vulnerability databases with the help of the tool.

4

Accurate and Scalable Cross-Architecture Cross-OS Binary Code Search with Emulation

4.1 Introduction

With the prosperity of open source projects and open software under GPL, new software (even commercial software) can reuse the code of the existing related projects. However, extensive reuse of existing code leads to some legal and security issues. According to the report on ‘gpl-violations.org’ [90], over 200 cases of GNU public license violations have been identified in software products.

Among them, VMWARE is the well-known software vendor that is facing the lawsuit regarding the GPL violation [91]. Meanwhile, code reuse brings risks that a vulnerability disclosed in reused projects can spread into the new commercial products as undisclosed vulnerability. For example, in Dec. 2014, vulnerabilities in the implementation of the network time protocol (NTP) *ntpd* affected the products that import it, such as Linux distribution, Cisco's 5900x and Apple's OSX switches.

When the source code of an executable or a library is not available, binary code search can facilitate the tasks of GPL violation detection [92], software plagiarism detection [1, 93], reverse engineering [94], semantic recovery [95], malware detection [96] and buggy (vulnerable) code identification [97–99] in various software components. However, binary code search is a fundamental problem that is much more challenging than the problem of source code search, due to the syntax gaps caused by the platform, the architecture and the compilation options.

In source code search, the similarity of two code segments is measured based on some representations of source code, e.g., approaches based on token [100], abstract syntax tree (AST) [101], control flow graph (CFG) [102], or program dependency graph (PDG) [103]. All these representations capture the syntactic or structural information of the program, and yield accurate results for source code search. However, these approaches fail when applied to binary code search. The reason is that the same source code may be compiled into the assembly code with different structures due to the choice of architecture, platform or compilation options. Thus, the syntactic or structural information is not sufficient for accurate binary code search.

In reality, binary code clone search is more complicated than the typical binary code clone search problem, which is finding binary code clones among two similar projects within the same architecture and platform. Usually, it needs to be cross-architecture and cross-OS. Imagine that a famous open-source project (e.g., libXML2) is reported to contain some critical buggy code, we need to find the corresponding buggy binary code of applications that reuse code of this project

on different architectures and platforms. To tackle the above problem, we propose three desirable characteristics for an accurate yet scalable cross-architecture and cross-OS binary code search tool.

- **P1.** Resilient to the syntactic and structural differences due to different configurations of compilers, architectures and OSs.
- **P2.** Accurate in capturing the abstract and complete function semantics, balancing the impact of compiler optimization levels.
- **P3.** Scalable to large-size real-world binaries, avoiding the overheads of the analysis based on dynamic executions or constraint solving.

The existing binary code search approaches adopt two types of analysis: namely, static analysis [1, 97, 98, 104] and dynamic analysis [105–108]. Static approaches usually rely on the syntactical and structural information of binaries, especially CFG (i.e., a control flow of basic blocks within a function) to perform the matching. Dynamic approaches often inspect the invariants of input-output or intermediate values of program at runtime to check the equivalence of binary programs. In general, static approaches are good at P3, but suffer from P1 and P2. On the contrary, dynamic methods have merits in P2 and P3, while having drawbacks in P3.

Table 4.1: Comparison of existing techniques. Here, ✓ or ✗ represent whether it supports the corresponding feature or not, respectively.

Tool	Type	Low-level semantic features	High-level semantic features	Structural features	Syntactic features	Constraint solving	Fast filtering	Selective inlining	Emulation
BLEX [108]	Dynamic	✓	✓	✗	✗	✗	✗	✗	✗
TRACY [98]	Static	✗	✗	✓	✓	✗	✗	✗	✗
DISCOVRE [109]	Static	✗	✗	✗	✗	✗	✓	✗	✗
CoP [1, 93]	Static	✓	✗	✗	✗	✓	✗	✗	✗
Bug search [97]	Static	✓	✗	✗	✗	✓	✗	✗	✗
Esh [110]	Static	✗	✗	✓	✗	✗	✗	✗	✗
BINGO [111]	Static	✓	✗	✗	✗	✓	✓	✓	✗
BINGO-E	Static	✓	✓	✓	✗	✗	✗	✓	✓

In Table 4.1, we list the state-of-the-art binary code search tools in the literature. Among these tools, only BLEX [108] is the dynamic function matching tool that uses seven semantic features generated from running execution. All the other tools adopt static analysis, and hence suffer from the issues in P1 and P2. TRACY [98] uses the pure syntax information for function matching, and it uses k -tracelet (i.e., basic blocks of length k along the control-flow path), which is architecture- and OS-dependent. DISCOVRE [109] proposes to find across-architecture bugs in binaries in a scalable manner, where it uses two filters (numeric and structural) to quickly locate the similar candidates for the target function. CoP [1] is a plagiarism detection tool that adopts the theorem prover to search for semantically equivalent code segments. The bug search tool proposed in [97] supports cross-architecture analysis by translating the binary code to an intermediate representation, and solving the resulting assignment formulas of input and output variables. ESH [110], inspired by the idea of similarity by the composition for images, proposes to decompose each procedure to small code segments (so called *strands*), semantically compare the strands to identify similarity, and lift the results into procedures.

In Table 4.1, we list the major features¹ used for similarity scoring and the search optimization techniques in these tools. *Low-level semantic fea.* refer to the features of low level architectural information used for matching (e.g., symbolic analysis on the CPU flag and register values [111]). *High-level semantic fea.* refer to the features of API-relevant information used for matching (e.g., the names and sequence of invoked built-in APIs of a lib [108]). *Structural fea.* refer to the structural information used by the static approaches (e.g., basic-block structures used in [1, 97, 109] and fixed length tracelet in [98]). *Syntactic fea.* refer to the syntactic information used for matching (e.g., the function name, or string literals extracted from binary [92]).

In general, merely relying on structural and syntactic features cannot make the match across the boundary of architecture, OS or compiler options, and hence fail P1. Instead, semantic features help to achieve P1. Even with the semantic features, the existing static approaches may not capture the complete function

¹Refer to section 4.4 for details of each category of features.

semantics due to different function inlining options of candidate functions [97]. Hence, the proper inlining strategy is desirable for achieving P2. Besides, scalability is a real challenge for tools [1, 97] based on symbolic analysis (low-level semantic features) and constraint solving. For example, CoP took half a day to find target functions in Firefox for a few signature functions [1]. Thus, to achieve P3, time-consuming constraint solving should be avoided.

In our previous work [111], we propose a binary code search tool, named BINGO. For P2, BINGO adopts a *selective inlining* (§4.5) to include relevant libraries and user-defined functions in order to capture the complete semantics of the function. For P3, a filtering technique is proposed to shortlist the candidate functions. Subsequently, after inlining and filtering, for P1, BINGO extracts the partial traces of various lengths from the candidate functions as function models, and then extracts *low-level semantic features* from the function models (§4.4.3). Last, BINGO measures the function similarity score by Jaccard containment similarity of features.

In our previous evaluation on a number of real-world binaries [111], BINGO outperforms the available state-of-the-art tools (i.e., TRACY [98] and BINDIFF [112]), for the same tasks in terms of search accuracy and performance. However, we found some false positive (FP) cases in using BINGO. When a function has a small size of assembly instructions, the low-level semantic features extracted from this function could be too generic to contain any real semantics. For example, Fig. 4.2 shows a false positive case of BINGO due to the same input/output values (see details in §4.3.2).

To address the FP cases, the new version of our tool, BINGO-E, supports high-level semantic features and structural features. High-level semantic features we use are the information of system calls or library calls (§4.4.2). The rationale is that such information is commonly used for malware detection [113], as they can reveal the semantics of the functions. On the other hand, we borrow the idea of fast bytecode clone detection based on the approach of projecting the CFG into a centroid of 3 dimension coordinate [114]. Inspired by this approach (3D-CFG for short [114]), the problem of graph isomorphism is reduced to a problem of

calculating the distances among centroids. 3D-CFG implemented in BINGO-E adopts the following structural information (§4.4.1): basic block (BB) sequence, loop information, and in-degree/out-degree of BB.

To speed up the process, BINGO-E leverages on emulation rather than constraint solving that is used in extracting low-level semantic features. We adopt UNICORN [115], a lightweight multi-platform, multi-architecture CPU emulator framework, to virtually execute the partial traces that are extracted from the function models of a given function. The emulation step may take significantly less time than constraint solving. The challenge of integrating UNICORN [115] lies in the handle of function calls (§4.4.3.3). Last, similarity scores of two functions in terms of structures, low-level/high-level semantics are combined in the final matching (§4.6).

Key contributions. Beyond the contributions that have been made in our previous work [111], we make these new contributions:

- Our work is the first attempt to incorporate emulation with cross-architecture cross-OS binary code search.
- We extract structural features based on the idea of centroid based clone detection. Previously, 3D-CFG based matching was mainly applied for byte-code clone detection.
- We combine different categories of features listed in Table 4.1. We evaluate our approach under the scenarios of cross-architecture matching, cross-OS matching, and cross-compiler matching.
- We empirically compare BINGO-E with BINGO and the available state-of-the-art tools of binary code search in terms of search accuracy and performance.

Experimental results. We conduct various experiments to evaluate the effectiveness of BINGO-E. For cross-architecture matching on `coreutils` binaries, on average, the rate of best match for BINGO-E (65.6%) is significantly better than

that of BINGO (35.1%) . For cross-compiler matching and intra-compiler matching on `coreutils` binaries, on average, the rate of best matching is improved from the range of 30%-60% (for BINGO) to the range of 70%-99% (for BINGO-E). For cross-OS matching on Windows binary `mscvrt` and Linux binary `libc`, on average, the rate of best matching is improved from 22% (for BINGO) to 51.7% (for BINGO-E). In additional experiments on binary code matching on forked projects, BINGO-E achieves a higher rate of best matching (93.6%) than that (88%) of the existing benchmark tool. Last, by virtue of emulation, the most time-consuming part of low-level semantic feature extraction is significantly accelerated, which allows us to introduce more features to improve the accuracy.

4.2 Background and Motivation

In this section, we introduce the background knowledge, and state the challenges in the current research of cross-architecture and cross-OS binary code search with illustrative examples.

4.2.1 Background

In general, binary code clone search is not an easy task, more complicated than source code clone search. The reasons are listed as follow: 1). the same source code will compile into different instruction sets on different architectures, e.g., x86, ARM, SPARC, etc. 2). the same source code will compile into binary code with different control flow structures due to different compilers or compilation options, even on the same architecture and platform. 3). the same source code will compile into different binary code due to the inlining of different system library code on different OSes. An important concept is *function inlining* during compilation, which means that the compiler inserts the function code at the address of each function call, thereby saving the overhead of a function call. Generally, different compilation options adopt different inlining strategies for various function calls.

Hence, to mitigate the issue, we can apply the same strategy to *selectively inline* function calls before matching (see §4.5 how we selectively do it). In such way, two cloned binary functions, which originally have different function call inlining strategies during compilation, can be rightly matched.

To match binary code clones, two basic approaches are employed. The first one is to conduct *static analysis* by using tools like IDA PRO [116], DYNINST [117], and ANGR [118]. These tools allow to build the control-flow graph (CFG) of a binary function inside the binary program. Based on the similarity of CFGs, the similarity of two binary functions is measured. Recently, to match the bytecode clone on Android, the concept of 3D-CFG is proposed in [119]. 3D-CFG, as its name, projects the CFG of a function into a 3D-coordinate system. Traditionally, CFG similarity is measured by graph matching or graph isomorphism. In 3D-CFG, each function is considered as an object in the 3D-coordinate system, and two functions' similarity is measured by the distance between the mass centroids of the two objects that represent the two functions, respectively. The other approach is to conduct *dynamic analysis* by leveraging the instrumentation or emulation tools. Among these tools, the most fundamental one is QEMU [120], which is a free and open-source hosted hypervisor that performs hardware virtualization. QEMU can emulate most functionality of a computer. Many of modern emulators are extended on QEMU. Among these extensions, UNICORN [115] is a lightweight multi-platform, multi-architecture CPU emulator. It has high performance by using Just-In-Time compiler technique, and supports fine-grained instrumentation at various levels. Using UNICORN to execute binary functions, matching two binary code snippets can be reduced to comparing I/O values of two binary code snippets.

Figure 4.1: Code segment responsible for Heartbleed vulnerability (CVE-2014-0160) appeared as in the binary (a) compiled with gcc 4.6, and (b) compiled with mingw32

4.2.2 Motivating Example

A binary program consists of a number of functions, each of which can be represented by the CFG (control-flow graph), a (directed) graph of basic blocks (BBs). The assembly instructions in a function are systematically grouped into several BBs, which are considered as the building blocks of binary program. Thus, this BB-based representation is used by many static binary analysis tools.

The same source code may produce binary code of different BB structures, due to different compilation configurations. Taking the heartbleed vulnerability (CVE-2014-0160) for example, Fig. 4.1(a) and (b) show the BB structures of the same source code compiled with `gcc` and `mingw`, respectively. Apparently, these two binary code segments share no identical BB structures — with `gcc`, the vulnerable code is represented as a single basic block; with `mingw`, represented as several basic blocks. A detailed inspection suggests that library function `memcpy` is inlined in `mingw` version; while in `gcc`, it is not. Huge differences between the binaries in syntax and program structure make binary code search a challenging task.

4.2.3 Challenges for Syntactic Approaches

Syntax is the most direct information usable for code search. Most of existing approaches that rely on syntax information have attempted to use instruction patterns [98, 104, 121, 122]. As no consistent low-level syntax representation (i.e., assembly instructions) is available for cross-architecture search, these approaches fail for cross-architecture analysis. To make binary code search across the architecture and OS boundary, semantics-based matching has been proposed [1, 97], in which the machine state transition represents the semantics of a binary. Still, three challenges are faced in semantics-based matching.

4.2.4 Challenges for Semantics-based Matching

Challenge 1 (C1): The trade-off challenge of deciding the granularity of function model. The state-of-the-art tools CoP [1] and bug search [97] assume that BB-structure is preserved across binaries, and a single BB can be matched with other BBs of similar semantics. Based on BB-structure, semantic features are extracted and built into the function model. However, as admitted by Pewny *et al.* [97], this approach is too sensitive to BB-structure differences, and hence problematic for smaller functions whose CFG structures are more susceptible to compilation options. For example, the BB-structure compiled with `gcc` 4.6 in Fig. 4.1(a) is significantly different from that compiled with `mingw32` in Fig. 4.1(b). In this case, the approach of BB-centric matching fails as no two BBs can match in Fig. 4.1.

Challenge 2 (C2). The accuracy challenge of using low-level semantic features. Functions are considered in isolation by most of static tools, i.e., semantics of callee functions are not considered as part of the caller's semantics. This leads to partial semantics problem, especially when some common utility functions are implemented by the developers themselves (e.g., Adobe Reader has its own `malloc` implementation). *Blindly inlining* all the callee functions can be a remedy, as all the user-defined functions are inlined in [123]. Nevertheless, this approach fails in practice due to two main reasons: (1) heavy inlining may lead to code size explosion [124], and (2) not all the callee functions are semantically relevant to the caller function. Thus, an inlining strategy is needed to decide that: `memcpy` should be inlined in Fig. 4.1(a) for matching with the semantically relevant one in Fig. 4.1(b). On the other side, for functions not to be inlined, e.g., `ss13_write_bytes` in Fig. 4.1(a), features of library call information should not be ignored.

Challenge 3 (C3). The performance challenge of using static symbolic analysis or dynamic execution. Syntax based techniques, in general, are scalable [98]. As discussed earlier, they fail on cross-architecture analysis. To address this problem, semantics based approaches are preferred. However, extracting

Figure 4.2: A false positive case of BINGO

and solving low-level semantic features is a time-consuming job. Previously, we adopted an *efficient* and architecture-, OS- and compiler-*neutral* function filtering mechanism in [111]. With such mechanism, extracting input/output values of registers and flags of a function via constraint solving still takes 4,469s on average to extract low-level semantic features from 2,611 Linux `libc` functions — it takes 1.7s to extract semantic features from a `libc` function. Hence, this part still has much space for improvement. To scale up code search for large-size binaries in real world, an efficient method is desired.

Previously, BINGO has adopted *k*-tracelet for C1, selective inlining for C2, and filtering mechanism for C3 [111]. However, BINGO still suffers issues of false positive cases (see Fig. 4.2 and §4.3.2) and inefficient feature extraction. In this chapter, BINGO-E adopts new features and emulation to address these issues (see §4.4).

4.3 System Overview of BinGo-E

BINGO-E, as a new extension of BINGO, aims to be an accurate and scalable search engine for binary code. Given a binary function to be matched (*signature function* in search), BINGO-E returns the functions from the pool of analyzed binaries (*target function* in search), ranked based on their semantic similarity.

Figure 4.3: BINGO-E work flow: rectangles represent processing steps, ovals represent features, arrows represent control/data flows

4.3.1 The Sketch of Work Flow

Fig. 4.3 shows the work flow of BINGO-E. The input is a binary function as the signature function and a set of candidate functions as possible targets. The output is the ranked list of candidate target functions above the user-defined threshold of similarity score. The basic idea is the two-phase matching: first using the structural and high-level semantic features to find similar target functions; if the similarity scores of all candidate target functions are below a certain threshold (*No* flow after primary matching in Fig. 4.3), the low-level semantic features are used to compare I/O values on k -tracelets of the signature function with those of candidate target functions.

We explain how each step in our approach works in brief. At step 1, given a signature function, it performs the preprocessing, i.e., disassembling and building the CFG for the signature function. At step 2, we extract these features from the high-level semantics and structure information in Table 4.2. As shown in Table 4.2, we extract 6 types of high-level semantic features² (§4.4.2) and 3D-CFG structural features (§4.4.1). At step 3, a primary matching is conducted

²We do not claim that the 6 types of features are complete features for the purpose of binary matching. We chose the set of features based on the related literature and feasibility of our approach. Furthermore, we argue that the 6 types of features are effective as demonstrated in the experiment section. Hence, we summarize the effective high-level features from literature review.

to measure the similarity based on the Jaccard distance of high-level semantic features and the centroid distance of structural features (§4.6.1). If no results are above the similarity threshold, the process proceeds with comparing low-level semantic features (§4.6.2).

The second-phase matching begins if step 3 returns no clones. At step 4, for each function in the target binaries, function calls are identified and selectively inlined according to the relevance of called libraries and other user-defined functions (§4.5). Note that selective inlining is for addressing C2 (§4.2.4). Selective inlining is transitive. Assume function a calls function b and b calls function c . If a selectively inlines b , then the decision of inlining c into b is according to Algorithm 3 (§4.5). If a does not need to inline b , then it is certainly unnecessary to inline c into b . At step 5, for the signature functions, length variant partial traces (up to k -tracelets) are generated and grouped to form the function models (§4.6.2.1), which low-level semantic features are extracted from (§4.6.2.2). Note that function model generation and feature extraction is a one-time job. At step 6, we rely on emulation to get the input/output values, flags, memory addresses of signature functions as low-level semantic features (§4.4.3.3). Finally, at step 7, combining Jaccard Distance for low-level semantic features and results of the primary matching, BINGO-E returns target functions that are above the similarity threshold in a ranked order (§4.6.3). Note that the low-level and high-level semantic features are some forms of signatures of a binary function, we apply Jaccard Distance [125] (see §4.6.1 for the detailed usage) to measure the similarity of two binary functions in terms of low-level and high-level semantic features. As the structural features are represented as 3D-CFG, the structural similarity is measured based on the distances between the weighted 3D-coordinates of two binary functions (see Definition 3). Finally, the overall similarity is the weighted aggregation of different categories of similarity values.

The approach is presented in the following way: in §4.4, different categories of features and emulation (step 2 and 6) are introduced; in §4.5, selective inlining (step 4) is elaborated; in §4.6, the key steps 3, 5 and 7 are explained with details.

Table 4.2: Different categories of features used by BINGO-E

Low-level Semantic Feature	High-level Semantic Feature	Structural Feature
Memory operations	Operation type	BB sequence
Flags	System call tags	Loop information
Registers	Function call sequence	In-degree of BB
	Function parameter	Out-degree of BB
	Local variable	
	Opcode	

4.3.2 Solutions to the Matching Challenges

Here, we highlight how BINGO-E overcomes the challenges aforementioned in §4.2.3 and improves BINGO.

S1: *Flexible function model via k -tracelet and 3D-CFG.* In BINGO and BINGO-E, to address C1, we borrow the idea of tracelet in TRACY [98], where David *et al.* extract a fixed-length partial trace of 3 BBs (3-tracelet). Then they check semantic equivalence between 3-tracelets based on data-dependence analysis and rewriting, in which constraining solving is applied to verify input/output invariants among the registers, flags and variables. In BINGO, we use a set of variant-length partial traces (i.e., a set of k -tracelet, k is up to 4) to represent the function model. In BINGO-E, we complement the tracelet-based approach with function structural information (i.e., 3D-CFG). 3D-CFG is resilient to BB structure changes due to optimization levels, as it uses structural information of a function as a whole.

S2: *Completing function semantics via *selective inlining* and identifying the *high-level semantics*.* To mitigate the issue of incomplete semantics of the analyzed function, selective inlining is proposed in BINGO. However, several FP cases are observed during semantic matching, where low-level semantic features are insufficient to reveal the true functionality. For instance, two code segments in Fig. 4.2 cannot be distinguished by low-level semantic features as they have

the same input and output results for any given string input. In fact, they have different functionalities, as the code in Fig. 4.2(a) copies the string array via calling function *strcpy*, and the code in Fig. 4.2(b) does the memory copy (not limited to string copy) in an iterative way. In BINGO-E, we consider not only low-level semantic features (e.g., symbolic features on register, flag, memory status), but high-level semantic features that are relevant to function calls, including system calls and library calls.

S3: Efficient semantics extraction via binary code *emulation*. In order to speed up the process of extracting low-level semantic features, we propose to apply Unicorn [115] to virtually execute the extracted partial traces and identify the input/output values from them. Unicorn is a lightweight multi-platform, multi-architecture CPU emulator framework, which is built on QEMU. According to the manual [115], for extracting low-level semantic features, emulation may boost the efficiency by one or two orders of magnitude, compared with constraint solving.

To sum up, in Fig. 4.3, step 2, 3, 6 and 7 are newly introduced into BINGO-E. Among them, step 2 and 3 are the implementation of **S1** and **S2**, while step 6 is the implementation of **S3** (§4.3.2). Note that in step 2, only the static analysis is required so that the process of feature extraction is scalable. In step 6, to avoid the overheads of actual execution or constraint solving for extracting low-level features, emulation is adopted to speed up the process. In step 7, to remove the false positive cases due to matching with low-level semantic features alone, the results of step 2 (similarity in high-level semantics and BB-structure) are also considered (§4.6.3).

4.4 Binary Code Matching Features

In this section, we elaborate the features that we adopt to measure the similarity of two binary code segments. As listed in Table 4.2, these features fall into three categories. For each feature in Table 4.2, the example and description are given in the following subsections. Features of structures and high-level semantics are

extracted via static analysis, which is the functionality of step 2. Features of low-level semantics are extracted by emulation (step 6).

4.4.1 Structural Features

3D-CFG [119] depicts the CFG structure of a function at a high-abstraction level. It is used to measure the similarity between methods in/among bytecode of Android apps. The basic idea in [119] is to assign a structural 3D-coordinate value for each node in the control flow graph (CFG) of a method, then calculate the mass centroid of these coordinates as the centroid of the method. By calculating the distance between two methods' centroids, the similarity among these two methods can be measured.

According to [119], for a given method, each node in the CFG is a basic code block. Each node has a unique coordinate, denoted as a 3-tuple (x, y, z) , where x is its sequential number in the CFG; y is the number of its outgoing edges (out-degree of the node in CFG), and z is the loop depth of the node.

Based on the nodes inside the function f , we define the centroid $\vec{c}_f$ of function f :

Definition 1. The centroid is a 4-tuple (c_x, c_y, c_z, w) [119]:

- $w = \sum_{e(p,q) \in 3D-CFG} (w_p + w_q),$
- $c_i = \frac{\sum_{e(p,q) \in 3D-CFG} (w_p i_p + w_q i_q)}{w}, \text{ where } i \in \{x, y, z\}$

where $e(p, q)$ is the edge between the node p and q , (x_p, y_p, z_p) is the coordinate of the node p , and w_p is the number of assembly instructions in the node of BB.

To emphasize the importance of function call instruction (relevant to high-level semantics), the weighted centroid is defined as $\vec{c}_f' = (c'_x, c'_y, c'_z, w')$ for function f , where $w' = w + N$, and N is the number of function call instructions; c'_x , c'_y and c'_z are recalculated according to the new weight w' , respectively. Suppose each BB node in Fig. 4.4 contains only one instruction ($w_A = 1, w_B = 1, w_D = 1$), and

node C contains one function call instruction ($w_C = 2$). We have the following coordinates: $(1, 2, 1)$ for A , $(2, 2, 1)$ for B , $(3, 1, 0)$ for C and $(4, 0, 0)$ for D . Then, the centroid is $(2.2, 1.5, 0.6, 10)$ and the weighted centroid is $(2.38, 1.38, 0.46, 13)$.

Given a function f , the function signature $\vec{f} = (\vec{c}_f, \vec{c}_f')$ is a feature vector containing the structural information of the function, where $\vec{c}_f$ is the centroid, $\vec{c}_f'$ is the weighted centroid.

The difference between the centroids of two functions f_1 and f_2 is the primary condition to evaluate their similarity:

Definition 2. The Centroid Difference Degree (CDD) [119] of two centroids $\vec{c}_1 = (c_{1x}, c_{1y}, c_{1z}, w_1)$, $\vec{c}_2 = (c_{2x}, c_{2y}, c_{2z}, w_2)$ (the distance of the weighted centroids $\vec{c}_1$ and $\vec{c}_2'$) is

$$CDD(\vec{c}_1, \vec{c}_2) = \max\left(\frac{|c_{1x} - c_{2x}|}{c_{1x} + c_{2x}}, \frac{|c_{1y} - c_{2y}|}{c_{1y} + c_{2y}}, \frac{|c_{1z} - c_{2z}|}{c_{1z} + c_{2z}}, \frac{|w_1 - w_2|}{w_1 + w_2}\right)$$

Given two functions, the function-level difference degree is defined as the maximum value between their centroid distance (CDD) and their weight centroid distance (weighted CDD):

Definition 3. Function Difference Degree (FDD) of two functions $\vec{f}_1 = (\vec{c}_1, \vec{c}_1')$ and $\vec{f}_2 = (\vec{c}_2, \vec{c}_2')$ is

$$FDD(\vec{f}_1, \vec{f}_2) = \max(CDD(\vec{c}_1, \vec{c}_2), CDD(\vec{c}_1', \vec{c}_2'))$$

The two code segments in Fig. 4.2 will not be matched as similar according to these structural features. Since the code segment in Fig. 4.2a has only one BB and the code segment in Fig. 4.2b has 3 BBs. Their CDD and weighted CDD are all 1 (c_{1y} and c_{1z} are all 0), which indicates zero similarity according to [119][126]. Thus, 3D-CFG captures the overall structure of a binary function; and it is effective in measuring the structural similarity of two given binary functions.

Note that the challenge of applying 3D-CFG for binary code search is inherent in the challenge and complexity of building the accurate CFG for binary code [127]. In commonsense, decompiling Java byte-code is easier than decompiling binary code. Hence, issues, e.g., getting the accurate loop (the indirect call for loop

Structural features:

BB sequence:

main: $A \rightarrow B \rightarrow C \rightarrow D$

the sequential number of BB (A): 1

In-degree of BB (A): 2

out-degree of BB (A): 2

Loop information:

The loop depth of BB (A): 1

involving number of BBs: 2

Figure 4.4: The BB-structure of a function, and its structural features

jump) structure and distinguishing data from code, need to be solved to further improve the accuracy of 3D-CFG. These issues are out of the scope of this work.

4.4.2 High-level Semantic Features

For the high-level semantics, we mainly make use of the operation types, system call tags (tags of function calls to system APIs), function call sequences, function call parameters, local variables and op. code. The basic idea is that function calls (especially system calls) carry the semantics of code and they can be used for semantic-based code matching tasks, e.g., system-call based malware detection [113]. Hence, the information of function calls (tags, parameters, sequences) is extracted as features. Additionally, we extract features from the numbers of different types of binary instructions. We first classify different binary instructions into 15 categories³, including *data_transfer*, *arithmetic*, *stack*, *logical*, *shift_rotate*, *control_transfer*, *loop*, *string*, *flag*, *misc*, *sign*, *fstp*, *port*, *mmx*, *call*. Then we use the number of instructions in each category as a feature. Last, we also extract features from local variables. The reason is that local variables may represent the intermediate values in the execution, and the same intermediate values indicate the same or similar function semantics.

³The instructions inside each category can be found from this link: <https://sites.google.com/site/toolbingoe/instruction-category>

In Fig. 4.5, we give an example for the high-level semantic features. In Fig. 4.5(e), the semantics of the function *foo2* is to do the string copy. Its binary code contains three BBs, in which B_1 has two out-edges to B_2 and B_3 , and B_3 has one out-edge to B_2 . The high-level semantic features in Fig. 4.5(d) are extracted from all the BBs of the function. Among all the instructions, we have 13 instructions of *data_transfer*, 10 of *stack*, 4 of *call*, 1 of *control_transfer*, and so on. Similar to the idea of mapping concrete system calls into abstract actions in [113], we add tags to the common system calls or library calls in order to capture the high-level semantics of the function. For example, the tag of function call *scanf* is *io*, the tag of function call *strlen* is *string*, and the tag of *memcpy* and *memset* is *mem*⁴. Similar to the idea of using API usage patterns for code recommendation [128] (source code search), we also extract the function call sequences on the longest program path: *scanf* $\rightarrow$ *strlen* $\rightarrow$ *memcpy* $\rightarrow$ *memset* on the control flow $BB_1 \rightarrow BB_3 \rightarrow BB_2$. Function *foo2* takes no parameters, and has the value of *null* for this feature. Local variables used in source code is *len*, which stores the return value of function call *strlen* in Fig. 4.5(e). However, in the binary code (B_1 in Fig. 4.5(a)), the address at *ebp* – 12 stores the return value of function call *strlen* (according to the instruction “push *eax*” that stores the return value of “call *strlen*” into *eax* and the instruction “mov DWORD PTR[*ebp* – 12], *eax*”). So we use the memory address *ebp* – 12 as the feature for local variable. Last, the instruction types for all instructions in all BBs are used as a feature.

Note that inside these high-level semantic features, we make two minor technical improvements. First, to tag the instructions according to their hidden semantics, the binary code instructions are grouped into 15 categories, which are defined according to our domain knowledge and the relevant study (e.g., taint tagging in the study [129]). Second, to tag the system calls according to their hidden semantics, we also group system calls on Windows and Linux into various *actions*, which are widely used in malware detection but not yet in binary code search. *Action* means a group of semantically similar system calls [113]). Via these two minor improvements, we can better capture the high-level semantics.

⁴A more complete mapping list for function call tagging can be found from this link: <https://sites.google.com/site/toolbingoe/system-call-tags>

The two code segments of the false positive case in Fig. 4.2 is not matched as similar according to these high-level features. Code segment in Fig. 4.2a has function call, while code segment in Fig. 4.2b does not. They also have different local variables and opcode. For example, local variables are `ebp - 12` and `ebp - 16` in Fig. 4.2a, while they are `ebp - 4` and `ebp - 8` in Fig. 4.2b. Therefore, high-level semantic features are very helpful in recovering function semantics.

4.4.3 Low-level Semantic Features

The basic idea of capturing low-level semantic features is to compare the input/output pair [1][97], or compare the frequencies of different types of memory operations [108]. In BINGO, features of input/output pairs are adopted for comparison (§ 4.4.3.1). In BINGO-E, as emulation is supported to improve the efficiency, we extend the approach to support features that are based on values (including intermediate values) read from (or written to) the program stack and heap (§ 4.4.3.2).

4.4.3.1 Features Used in BinGo

Similar to [1] and [97], a set of symbolic formulas (called *symbolic expressions*) are extracted from each partial trace, where they capture the effects of executing the partial trace on the machine state. Here, machine state is characterized by a 3-tuple $\langle mem, reg, flag \rangle$ denoting the memory *mem*, general-purpose registers *reg* and the condition-code flags *flag* [1, 97]. For example, a machine state, before executing a partial trace, is given by $\mathcal{X} = \langle mem, reg, flag \rangle$, then just immediately after execution, the machine state is given by $\mathcal{X}' = \langle mem', reg', flag' \rangle$. Here, $\mathcal{X}$ and $\mathcal{X}'$ are referred to as *pre-* and *post-state*, respectively, and the symbolic expressions capture the relationship between these pre- and post-machine states.

In order to measure the similarity between the signature function and a target function, one can use a constraint solver, such as Z3 [130], to calculate the semantic *equivalence* between the symbolic expressions extracted from the signature and target functions. However, using a constraint solver to measure semantic equivalence is very costly [1] and not scalable for real-world cases, considering the complexity of pair-wise matching between the signature function and a possibly huge number of target functions. To

tackle the scalability issue, in [97], a machine learning technique is applied. Specifically, Input/Output (or I/O) samples are randomly generated from the symbolic expressions and are fed into the machine learning module to find semantically similar functions. Here, I/O samples are the input and output pairs — assigning concrete values to the pre-state variables in the symbolic expressions and obtaining the output values of the post-state variables.

One of the drawbacks in randomly generating I/O samples is that the dependency among the symbolic expressions is ignored [97]. For example, consider the following 2 symbolic expressions:

$$\mathbf{zf}' \equiv (\mathbf{edx} < 2) \wedge (\mathbf{edx} > 0) \quad (4.1)$$

$$\mathbf{ecx}' \equiv \mathbf{edx} + 4 \quad (4.2)$$

where symbolic expression 1 (Eq. 4.1) has one pre-state (i.e., $\mathbf{edx}$) and one post-state variable (i.e., $\mathbf{zf}'$), similarly, the symbolic expression 2 (Eq. 4.2) also has one pre- (i.e., $\mathbf{edx}$) and one post- (i.e., $\mathbf{ecx}'$) state variable. Thus these two symbolic expressions are mutually dependent through a common pre-state variable $\mathbf{edx}$. However, in randomly generated I/O samples (as in [97]), $\mathbf{edx}$ can be assigned to two different values in each symbolic expression, which ignores the dependency and may lead to inaccurate semantics.

In the above example, to satisfy the mutual dependency on pre-state $\mathbf{edx}$, $\mathbf{edx}$ must be 1 and $\mathbf{ecx}'$ must be 5 to ensure $\mathbf{zf}'$ to be **true**. Hence, constraint solving is applied to find appropriate values for pre- and post-state variables, as it considers all the symbolic expressions from the same partial trace.

Note that in the above example, we compare the input/output value pair for the same registers, assuming they have the same initial state. However, for cross-architecture code search, input/output values of registers cannot be used, as no identical registers exist on different architectures. In BINGO, only input/output values of some flags and memory addresses are used. Thus, to further improve the accuracy, more features are needed for cross-architecture code search.

4.4.3.2 Features Newly Added in BinGo-E

As aforementioned, the input/output values of registers cannot be directly used for cross-architecture code search. Owing to emulation, BINGO-E is now much faster and more convenient at extracting features of function arguments and intermediate values used by the function. Hence, to further improve the accuracy of low-level semantic features, we propose to incorporate the following features of intermediate values that are also used in BLEX [108]:

1. Values read from the program heap (denoted as “v1”)
2. Values written to the program heap (denoted as “v2”)
3. Values read from the program stack (denoted as “v3”)
4. Values written to the program stack (denoted as “v4”)

According to the results reported by Egele *et al.* [108], features of “v1” can achieve a matching accuracy of 40%, and features of other values can achieve the accuracy between 50% ~ 60%. Hence, using these features together can help find code segments that own identical or similar function parameters and local variables (stack operations), intermediate objects (heap operations). Since we adopt emulation to extract the low-level semantic features, the above 4 types of features are extracted via the address analysis during the emulation of the assembly instructions.

The address analysis is based on the basic idea: before the emulation, we assume the starting address of the stack (**ebp**) as a certain hex value. Then in the emulation, we keep tracking the end address of the stack (**esp**) after each instruction is executed. If an operation falls in the address between **ebp** and **esp**, we can know the operation is on stack (belonging to “v3” or “v4”). Otherwise, we will consider it as the operation on heap (belonging to “v1” or “v2”). However, there are few cases where the instructions are operated on the global stack (the stack for global variables). In emulation, it is difficult to further distinguish those non-heap addresses, and hence we just consider these addresses as heap addresses.

4.4.3.3 Emulation

In BINGO, for a code segment or a k -length partial trace (i.e., k -tracelet), the low-level semantic features (e.g., the symbolic expression in § 4.4.3) are extracted and constraint solving is applied to resolve these symbolic expressions. Executing code segments in BLEX [108] or solving them by SMT solver in BINGO lead to an intractable problem — the scalability problem especially when the code segment is long and the symbolic analysis becomes complicated .

Emulation is faster than actual execution and much faster than constraint solving. In BINGO-E, emulation is adopted for extraction of feature values introduced in § 4.4.3.1 and § 4.4.3.2. Given two code segments, we assume they have the same initial memory status (i.e., the same address and layout). For code matching on the same architecture, we check the values of registers, flags after emulation and the addresses of read/write operation on stack and heap during emulation; while for code matching across architectures, we check the values of flags after emulation and the addresses of read/write operation on stack and heap during emulation. For example, for three code segments in Fig. 4.6, with the same initial memory status, they have the same resulting memory status — the overflow flag (V-flag) is 0, the zero flag (Z-flag) is 1, the negative flag (N-flag) is 0, and the carry flag (C-flag) is consistent (0 for x86 in Fig. 4.6(a) and (b) has the same meaning as 1 for Arm in Fig. 4.6(c)). Being code matching on the same architecture, code segments in Fig. 4.6(a) and (b) even have the same resulting values for registers `eax` and `ebx`. In addition, we also use the low-level features on values read from or written to the program heap/stack in this case. For example, `0x1100` is the intermediate value written to the stack.

In emulation part, the major technique challenges are 1). handling the function call and 2). choosing input or initial value for registers, flags and memory addresses. Note that emulation does not conflict with selective inlining. The emulation step is after the selective inlining step, and hence many important function calls (e.g., system calls and library calls) have been inlined. For function calls that are not selected for inlining, we adopt two different strategies for different scenarios:

1. In matching binary code from the same code base, we adopt the strategy of

inlining-none — that is none of these called functions will be inlined. The rationale is twofold: functions that are not inlined by selective inlining are less likely representing the semantics of the program (more potential of being utility functions); since signature function and target functions both have the called function (due to the same code base), for the same input, ignoring the called function should still produce the same results for the semantic clones.

2. In matching binary code from different code bases, we adopt the strategy of inlining-all, namely the strategy that all the called function will be inlined to emulate the execution of instructions. The reason is that binary segments compiled from different code bases usually call different system calls or library calls — ignoring them will bring errors and lead to different results for the same input, even if the two binary segments are semantic clones.

Regarding the emulation input, we adopt the strategy of using three totally different input values. For two binary functions to be matched via emulation, we feed it with three types of input: all values with `0x0000`, all values with `0x1111`, and all random values. If two binary segments are matched according to low-level semantic features for all three types of input, we considered they are matched.

```

push    ebp
mov     ebp, esp
sub     esp, 56
sub     esp, 8
lea     eax, [ebp - 16]
push    eax
push    OFFSET FLAT : .LC0
call    scanf
add     esp, 16
mov     eax, DWORD PTR[ebp - 16]
sub     esp, 12
push    eax
call    strlen
add     esp, 16
mov     DWORD PTR[ebp - 12], eax
cmp     DWORD PTR[ebp - 12], 9
jg      .L2

```

(a) basic block B_1

```

mov     eax, 0
leave
ret

```

(b) basic block B_2

```

mov     edx, DWORD PTR[ebp - 12]
mov     eax, DWORD PTR[ebp - 16]
sub     esp, 4
push    edx
push    eax
lea     eax, [ebp - 56]
push    eax
call    memcpy
add     esp, 16
mov     eax, 10
sub     eax, DWORD PTR[ebp - 12]
mov     ecx, eax
mov     eax, DWORD PTR[ebp - 12]
lea     edx, [0 + eax * 4]
lea     eax, [ebp - 56]
add     eax, edx
sub     esp, 4
push    ecx
push    32
push    eax
call    memset
add     esp, 16

```

(c) basic block B_3

Operation type:

data_transfer(13), *stack*(10),
arithmetic(11), *call*(4), *logical*(1),
control_transfer(1), *misc*(1),

System call tags:

io, *string*, *mem*, *mem*

Function call sequence:

scanf → *strlen* →
memcpy → *memset*

Function parameter: Null

Local variable:

ebp - 12(*mov*[*ebp* - 12], *eax*)

Opcode:

push → *mov* → ...
... → *leave* → *ret*

(d) High-level semantic features of B_1 , B_2 and B_3

```

int foo2()
{
    char *buf;
    char *p[10];
    scanf("%s", &buf);
    int len;
    len = strlen(buf);
    if(len < 10){
        memcpy(p, buf, len);
        memset(p + len, ' ', 10 - len);
    }
    return 0;
}

```

(e) the source code of these binary code segments

Figure 4.5: the high-level semantic features of an exemplar function

Figure 4.6: An example of using emulation to extract low-level features

Figure 4.7: Commonly observed function invocation patterns, where ‘UD’ denotes user-defined callee function. Here, all the incoming and outgoing edges (or calls) represent user-defined functions, unless specified as *Libcall* or *Termination Libcall* next to them

4.5 Accurate: Selective Inlining

Inlining is a technique in compilers to optimize the binaries for maximum speed or minimum size [124]. During compilation, the strategy of inlining is different according to the configured optimization level, which creates the challenge for binary code search. To mitigate this issue, we propose a selective inlining strategy which is based on function invocation patterns. Note that our strategy has a different goal in contrast with compilation process — ours is for program semantics recovery, while compilers' is for speed or size optimization.

4.5.1 Function Invocation Patterns

In order to inline relevant functions, we use the function invocation patterns to guide the inlining decision. Based on our study, six commonly-observed invocation patterns are identified and summarised in Fig. 4.7. Incoming (outgoing) edges in Fig. 4.7 represent the incoming (outgoing) calls to (from) the function. Here, we elaborate the six patterns as follows.

Case 1: Fig. 4.7(a) depicts the direct invocation of standard C library function(s) by the caller function under investigation. To recover the semantics, it is essential to understand the semantics of called library function(s). Hence, the library function is inlined into the caller function. Currently, we only consider the most common standard C library functions — in total 60 functions, from both Linux (`libc`) and Windows (`msvcrt`), for inlining. This list can be further extended when necessary.

Case 2: Fig. 4.7(b) depicts the case of a recursive relationship between the caller and the UD (user-defined) callee function f . Hence, we inline f into its caller. Note that the recursive functions are, unlike in compilers, inlined only once. For example, `gcc` has a default inlining depth of 8 for recursive functions.

Case 3: Fig. 4.7(c) depicts the common pattern of a *utility function* — e.g., the UD callee function f is called by many other UD functions, while f calls several *library functions* and a very few (or zero) UD functions. This suggests that f is behaving as a utility function, as f has some semantics that is commonly needed by other functions, and hence f is likely to be inlined.

Case 4: The UD callee function f in Fig. 4.7(d) is a variant of 4.7(c), where it has several references to library functions and *zero* reference to other UD functions. Such zero reference to UD functions makes f an ideal candidate for inlining. Note that f is inlined as the majority (50% or more) of its invoked library functions are not of termination type. Hence, we can safely assume that f is doing much more than just facilitating program termination. Here, termination type refers to the library functions that lead to exceptions or program termination (e.g., `exit` and `abort`).

Case 5: In Fig. 4.7(e), The UD callee function f is a variant of 4.7(d), which has only references to library functions. However, all the invoked library functions in f are of termination type. Hence, we consider as a function that facilitates only program termination (or exception handling) and its semantics are of little interest to the caller, which should not be inlined.

Case 6: Fig 4.7(f) depicts the scenario of a *dispatcher function* where the UD function f is called by (i.e., incoming calls) many other UD functions and f itself calls many other UD and library functions. In this case, f appears to be a dispatcher function without much unique semantics, and hence, in most cases, not inlined.

4.5.2 Inline Decision Algorithm

From the discussions above, in Fig. 4.7 (a), (b), (d) and (e) there is a clear criterion in deciding whether a callee should be inlined or not. However, for Fig. 4.7 (c) and (f), the most commonly observed invocation patterns, a systematic decision-making procedure is still needed. To identify the cases of commonly-used functions (i.e., utility functions) to be inlined, we borrow the *coupling* concept from the software quality and architecture recovery community. In software metrics, the coupling between two software packages is measured by the dependency between them, e.g., the software package instability metrics [131]. In this work, similarly, we measure the function coupling by the *function coupling score*.

Definition 4. Function Coupling Score refers to the complexity of the invocations involved in a function and is calculated as $\alpha = \lambda_e / (\lambda_e + \lambda_a)$, where λ_a represents the number of UD functions that refers to the callee, and λ_e represents the number of UD functions that is referred to by the callee.

Algorithm 3: Selective inlining algorithm

Data: caller $\mathcal{F}$, set of callee functions $\mathcal{C}$, set of termination lib. func. $\mathcal{L}_t$, set of inlining lib. func. $\mathcal{L}_s$
Result: inlined function $\mathcal{F}^I$

```

1 Algorithm SelectiveInline( $\mathcal{F}, \mathcal{C}, \mathcal{L}_s, \mathcal{L}_t$ )
2   foreach function  $f$  in  $\mathcal{C}$  do
3     // inline selected library functions
4     if  $f \in \mathcal{L}_s$  then
5        $\mathcal{F}^I \leftarrow \mathcal{F}.\text{inline}(f)$ 
6       return  $\mathcal{F}^I$ 
7     else if  $f \notin \mathcal{L}_s$  && isLibCall( $f$ ) then
8       return null
9     // for all other user-defined callee functions
10     $I_u^f \leftarrow \text{getIncomingCalls}(f)$ 
11     $O_u^f, O_l^f \leftarrow \text{getOutgoingCalls}(f)$ 
12     $O_u^f \leftarrow O_u^f \setminus \mathcal{F}$  // remove recursion
13    if  $|O_u^f| == 0$  &&  $(|O_l^f \cap \mathcal{L}_s| - |O_l^f \cap \mathcal{L}_t|) \leq 0$  then
14      return null
15    else
16       $\alpha = \lambda_e / (\lambda_e + \lambda_a)$  where  $\lambda_a = |I_u^f|$ ,  $\lambda_e = |O_u^f|$ 
17      // lower the  $\alpha$ , function  $f$  is likely to be inlined
18      if  $\alpha > \text{threshold } t$  && notRecursive( $\mathcal{F}, f$ ) then
19        return null
20      else
21         $\mathcal{F}^I \leftarrow \mathcal{F}.\text{inline}(f)$ 
22        if  $|O_u^f| > 0$  then
23          SelectiveInline( $f, O_u^f, \mathcal{L}_s, \mathcal{L}_t$ )
24        else
25          return  $\mathcal{F}^I$ 
26        end
27      end
28    end
29  end
30  return  $\mathcal{F}^I$ 

```

The lower the value of α , more likely the callee should be inlined. The rationale is that the low function coupling score implies the high independency of the functionality — the high possibility of being utility function. When calculating λ_e , we only consider the UD functions invoked by the callee, not the library functions. As mentioned in case 3 (Fig. 4.7(c)) and 6 (Fig. 4.7(f)), it is the invocations of UD functions that indicate the behavior of the callee as a dispatcher or a utility function, not the invocations of library functions.

The proposed selective inlining process is presented in Algorithm 3. As first, if the callee is one of the selected library functions (e.g., `memcpy` and `strlen` as in case 1), it is directly inlined into the caller (lines 3-5) and rest of the library functions are ignored. Next, the UD functions that refer to the callee is denoted as I_u^f ; while the library and UD functions referred to by the callee are identified and denoted as O_l^f and O_u^f , respectively at line 8-9. Callee that invokes only library functions that are of termination type is not inlined (lines 11-12). Finally, for the rest of the callee functions, the function coupling

score is calculated and if it is below some threshold value t , the callee is inlined (lines 13-24). This recursive procedure is continued until all the related functions are analysed.

In our preliminary study on BusyBox compiled for x86 32bit, we identify that 14 UD utility functions (case 3) have more than 50 incoming calls with 1 outgoing call, while 12 UD dispatcher functions (case 6) have more than 50 outgoing calls with just 1 incoming call. Similarly, in BusyBox compiled for ARM, we identify such 15 UD utility functions but only 4 UD dispatcher functions. This clearly differentiates the function invocation patterns case 3 (Fig. 4.7(c)) and 6 (Fig. 4.7(f)). In this work, we adopt a lightweight static analysis to make the inlining decision for the performance reason (as shown in Section 4.7). A more expensive program analysis can be used to improve the accuracy, but we strike for the balance between performance and accuracy in this work.

4.6 Scalable Function Matching

In BINGO and BINGO-E, similarity matching is done at the granularity of function and sub-function levels (i.e., we can match a part of a target function to the signature function).

4.6.1 Primary Matching

In primary matching, high-level semantic features and structural features are combined to measure the semantic similarity of the two code segments as the granularity of function. Let $\mathcal{SIM}_{\mathcal{H}}(sig, tar)$ denote the similarity score of using high-level semantic features and $\mathcal{SIM}_{\mathcal{S}}(sig, tar)$ denote the similarity score of using structural features. The former score is measured by Jaccard containment similarity [125], considering each function has a bag of high-level semantic features:

$$\mathcal{SIM}_{\mathcal{H}}(sig, tar) = \frac{\mathcal{H}_{sig} \cap \mathcal{H}_{tar}}{\mathcal{H}_{sig}} \quad (4.3)$$

Specifically, for each of six types of high-level semantic features, the Jaccard distance is calculated, and the overall $\mathcal{SIM}_{\mathcal{H}}(sig, tar)$ is the mean value of Jaccard distances of different types of high-level semantic features. Note that if both signature function

and target function have no function calls, then features in terms of function call tags, function call sequence, function parameters are not available, and they are not counted in overall similarity.

The latter score is measured by calculating FDD (Definition 3 in §4.4.1), and the FDD value is already normalized into the range between 0 to 1. $SLM_S(sig, tar)$ is to convert the distance to the similarity:

$$\begin{aligned} SLM_S(sig, tar) &= Convert(FDD(sig, tar)) \\ &= 1 - FDD(sig, tar) \end{aligned} \quad (4.4)$$

The idea of conversion is that if these two functions' distance is closer to 0, their similarity is closer to 1. Vice versa, if their distance is as big as it could possibly be, their similarity is prone to be 0.

Once we have similarity scores in high-level semantics and structures, the primary matching result $SLM'(sig, tar)$ is calculated as the weighted sum of $SLM_H(sig, tar)$ and $SLM_S(sig, tar)$, which is defined as follow:

$$\begin{aligned} SLM'(sig, tar) &= (1 - \mathcal{W}/2) * SLM_H(sig, tar) \\ &\quad + (\mathcal{W}/2) * SLM_S(sig, tar) \end{aligned} \quad (4.5)$$

where $\mathcal{W} = \min(\frac{w'_{sig}}{w'_{tar}}, \frac{w'_{tar}}{w'_{sig}})$ and $\mathcal{W}$ means the ratio of these two functions' instruction number.

Hence, the idea of primary matching is straightforward — structural features are effective when the sizes of two functions are similar. If two functions have the similar size of instructions, structural features have the same weight as high-level semantic features. If not, high-level semantic features have more weight than structural features. According to this idea, a high value of $SLM'(sig, tar)$ (e.g., ≥ 0.8) indicates that *sig* and *tar* are similar in both structures and high-level semantics.

4.6.2 Low-level Semantic Features Matching

When the primary matching does not find any ranked results that are of high $SLM'(sig, tar)$ values, low-level semantic features are used. Note that low-level semantics refer to not

Figure 4.8: A sample CFG (a) and the extracted 3-length (b) and 2-length (c) partial traces

only the implementation details, but also the partial semantics of the function. To this end, we propose the function model consisting of partial traces of various lengths, which is more flexible in terms of comparison granularity, compared with the BB-centric function modelling [1, 97]. Then, from these partial traces we extract symbolic expressions. Based on I/O values for register, flags and memory addresses, we match the partial traces inside two functions. Last, we apply Jaccard containment similarity [125] to measure the similarity score of two function models.

4.6.2.1 K-length Partial Trace Extraction

Our partial trace extraction is based on the technique proposed by David *et. al.* [98]. We omit the algorithm and explain the results using one example. Fig. 4.8 depicts a sample CFG of a function and the extracted 2- and 3-length partial traces (i.e., for $k = 2, 3$). We can observe that the original control-flow instructions (`jnb`, and `jb`) are omitted as the flow of execution is already determined. Note that the feasibility of the flow of executions is not considered at this step.

4.6.2.2 Function Model Generation

In BINGO, we leverage on length variant partial traces to model the functions. We generate partial traces with three different lengths (i.e., $k = 1, 2$ and 3) from each function, all of which collectively constitute the function model. For example, function

Figure 4.9: A sample signature (a) and target (b) functions, where the lines indicate the matched partial traces and the nodes refer to the basic-blocks.

models generated from the signature ($\mathcal{M}_{sig}$) and target ($\mathcal{M}_{tar}$) functions in Fig. 4.9 are listed as follows:

$$\mathcal{M}_{sig} : \{\langle 1 \rangle, \langle 2 \rangle, \dots, \langle 1, 2 \rangle, \langle 2, 5 \rangle, \dots, \langle 1, 2, 4 \rangle, \langle 2, 4, 7 \rangle, \dots\}$$

$$\mathcal{M}_{tar} : \{\langle c \rangle, \langle b \rangle, \dots, \langle a, b \rangle, \langle b, c \rangle, \dots, \langle a, b, c \rangle, \langle b, c, f \rangle, \dots\}$$

In BINGO and BINGO-E, we support the function model similarity matching in terms of *n-to-m*, *1-to-n*, *n-to-1* and *1-to-1* partial trace matching across the signature and target functions, mitigating the impact of program structural difference.

n-to-m Partial traces of length $n(\in \mathbb{Z}_{>1})$ generated from the signature function are matched against the partial traces of length $m(\in \mathbb{Z}_{>1})$ generated from target function. In Fig. 4.9, matching partial traces $\langle 3, 6, 8 \rangle$ and $\langle d, g \rangle$ is *3-to-2* matching.

1-to-n A basic-block (i.e., a partial trace of length 1) in the signature function is matched against the partial traces of length $n(\in \mathbb{Z}_{>1})$ generated from target function. In Fig. 4.9, partial traces $\langle 1 \rangle$ and $\langle 5 \rangle$ are matched with $\langle a, b \rangle$ and $\langle f, h, i \rangle$, respectively.

n-to-1 Partial traces of length $n(\in \mathbb{Z}_{>1})$ generated from the signature function are matched against a basic-block in the target function. In Fig. 4.9, partial trace $\langle 4, 7, 9 \rangle$ is matched with $\langle e \rangle$.

1-to-1 A basic-block in the signature function is matched against a basic-block in the target function. It is also known as pairwise comparison at the level of single basic-block. In Fig. 4.9, partial trace $\langle 2 \rangle$ is matched with $\langle c \rangle$.

1-to-n matching addresses the issue of basic-block splitting — a single basic block in the signature function is split into several smaller basic blocks in the target function. Similarly, *n-to-1* matching addresses the basic-block merging problem. In tracelet modeling [98], authors recommended that both the signature and target should be of the same size (i.e., $k = 3$), and hence only *n-to-n* matching is performed. In [97] and [1], pairwise comparison of single basic-block (i.e., *1-to-1* matching) is performed as an initial step to shortlist target functions. In contrast, in BINGO and BINGO-E, all the 4 types of function matching are needed to cover all possible BB-structure variances that arise due to architecture, OS and compiler differences.

For the similarity score of using low-level semantic features $\mathcal{SIM}_{\mathcal{L}}(sig, tar)$, considering the function model as a bag of partial traces, Jaccard containment similarity [125] is also used to measure the similarity of two different function models:

$$\mathcal{SIM}_{\mathcal{L}}(sig, tar) = \frac{\mathcal{M}_{sig} \cap \mathcal{M}_{tar}}{\mathcal{M}_{sig}} \quad (4.6)$$

where $\mathcal{M}_{sig}$ and $\mathcal{M}_{tar}$ refer to function models of low-level semantics that are generated from signature and target functions, respectively.

4.6.3 Final Matching

To remove false positives of using low-level semantics alone, $\mathcal{SIM}'(sig, tar)$ is also considered in the final matching.

Let $\mathcal{SIM}^*(sig, tar)$ denote the overall similarity score calculated in the final matching. $\mathcal{SIM}^*(sig, tar)$ is calculated as the weighted sum of $\mathcal{SIM}'(sig, tar)$ and $\mathcal{SIM}_{\mathcal{L}}(sig, tar)$. Here, we adopt two strategies for different matching scenarios.

If the matching is based on the same code base, we consider both $\mathcal{SIM}_{\mathcal{L}}(sig, tar)$ and $\mathcal{SIM}'(sig, tar)$ are equally important.

$$\begin{aligned} \mathcal{SIM}^*(sig, tar) = & (1/2) * \mathcal{SIM}'(sig, tar) \\ & + (1/2) * \mathcal{SIM}_{\mathcal{L}}(sig, tar) \end{aligned} \quad (4.7)$$

Otherwise, if the matching is based on different code bases (i.e., experiments on cross-OS matching), we ignore the structural features and give equal weight for low-level semantics and high-level semantics.

$$\begin{aligned} \mathcal{SIM}^*(sig, tar) = & (1/2) * \mathcal{SIM}_{\mathcal{H}}(sig, tar) \\ & + (1/2) * \mathcal{SIM}_{\mathcal{L}}(sig, tar) \end{aligned} \quad (4.8)$$

Still take the code segments in Fig. 4.2 as example. The two code segments are from different code bases, and hence the BB structure will not be similar. Low-level semantics and high-level semantics are used together to measure their semantic similarity. Previously, in BINGO, using low-level semantics alone suggests they have a similarity value of 0.7 — 70% of tracelets in the signature function match the target function on I/O value pairs. Now, in BINGO-E, since they have a low high-level similarity of 0.7 and no similarity in high-level features and structural features, their overall similarity is lower to 0.35. Hence, the false positive has a lower similarity score, and will be removed from the best-match or top-5-match list.

4.7 Experimental Results

Implementation. In BINGO and BINGO-E, we use IDA Pro to disassemble and generate CFGs from the functions in binaries. Partial traces are generated using the TRACY plugin⁵, where they are of three different lengths (i.e., k -tracelet with $k = 1, 2$ and 3). In BINGO, to facilitate analysis and feature extraction, similar to the preprocessing in [97] and [1], we lift the assembly instructions to an architecture and OS independent intermediate language, REIL [132]. In BINGO-E, no intermediate language is necessary as

⁵<https://github.com/tech-srl/tracy>

constraint solving based on REIL is not needed. All the experiments were conducted on a machine with Intel Core i7-4702MQ@2.2GHz with 32GB DDR3-RAM.

Configuration. For cross-architecture matching, experiments are conducted on the different executables (i.e., x86-32, x86-64, ARM version) of `coreutils`. For cross-compiler matching, experiments are conducted on the executables that are compiled from `coreutils` with the different combinations of adopted compilers (`clang` and `gcc`) and optimization levels (O0-O3). For cross-OS matching, we are matching between functions between `msvcrt` (for Windows) and `libc` (for Linux), which are compiled from different code bases with `gcc` for x86-32. For these experiments, we previously conduct them to evaluate BINGO; we also conduct them to evaluate how BINGO-E improves. To evaluate the real-world application of BINGO, we use the code of open source project vulnerabilities as signature to search for the similar vulnerability in commercial products (e.g., Adobe PDF Reader). For BINGO-E, we search for similar functions in binaries of forked projects, and compare the results with CoP [1]. Note that the term *query* used in some code search studies is referred to as *the signature function*. We are using the binary code of the signature functions as queries to search for the target functions.

4.7.1 Research Question

Through our experiments, compared with the results of BINGO, we aim at answering these four research questions (RQs):

1. How robust is BINGO-E in detecting *semantically equivalent* functions *across architectures* with code optimization?
2. How robust is BINGO-E in detecting *semantically equivalent* functions *across compilers* with code optimization?
3. How robust is BINGO-E in identifying *semantically similar* functions, across OS, in *wild* binary executables?
4. What are the key real-world applications of BINGO-E?

5. How does BINGO-E perform at the typical code clone search problem, compared with BINDIFF and TRACY?
6. How scalable is BINGO-E?

These RQs fall into three major topics, namely *robustness* (RQ1-3), *application* (RQ4) and *scalability* (RQ5). The robustness of BINGO-E lies in two aspects: (1) matching *semantically equivalent* functions, and (2) matching *semantically similar* functions. RQ1-2 are about the first aspect as we aim to match all functions in binary A to all functions in binary B , where the same source code (i.e., syntactically identical at the source code level) is compiled for different architectures using different compilers and code optimization options. RQ3 focuses more on finding semantically similar functions across binaries that are not compiled from the same source code (i.e., syntactically different even at the source-code level). In RQ1-2, we evaluate the selective inlining algorithm to show its effectiveness. In RQ3, we compare BINGO-E with BINGO, BINDIFF [112] and TRACY [98], indirectly showing the merits of incorporating different categories of features. In RQ4, we evaluate BINGO in finding real-world vulnerabilities and compare it against DISCOVRE [109] and [97], and evaluate BINGO-E in matching binary functions of forked open-source projects. In RQ5, we evaluate BINGO-E's effectiveness on solving the typical binary code clone detection under the same architecture, platform and compilation options, and compare it with BINDIFF and TRACY. Finally, in RQ6, we demonstrate the performance of BINGO and BINGO-E.

4.7.2 Answer to RQ1: Robustness in Cross-architecture Matching

4.7.2.1 BinGo's Answer to RQ1

We conduct two experiments. First experiment is to match all functions in `coreutils` binaries compiled for one architecture (i.e., x86 32bit, x86 64bit and ARM) to the semantically equivalent functions in binaries compiled for another architecture. For example, binaries compiled for ARM are matched against the binaries compiled for x86 32bit and x86 64bit architectures, and *vice versa*. In all three architectures, the

Figure 4.10: In BINGO, cross-architecture function matching before/after selective inlining using `coreutils` binaries. Here, **C** and **G** represent compilers `clang` and `gcc`, respectively, and 32 and 64 denote x86 32bit and 64bit architectures.

compilation uses `gcc` (v4.8.2) and `clang` (v3.0) with the optimization level `O2` (default settings in many Linux distributions).

Fig. 4.10 summarizes the average results (before and after selective inlining) obtained for `coreutils` binaries, where we report the percentage of functions ranked one (i.e., best match). In the plot, the bar **G64-G32** (after inlining) should be read as, when functions compiled, using `gcc`, for x86 64bit architecture are used as signature, around 55% of the functions compiled, using `gcc`, for x86 32bit architecture achieve rank one. It is observed that when ARM binary functions are used as signatures, the overall results significantly degrade — only 18% of the functions are ranked one, compare to x86 32bit and 64bit architectures, where it is around 41%. We also notice that matching(after inlining) between binaries compiled for ARM and x86 64bit yields higher ranking compared to ARM and x86 32bit binaries. The rationale is that ARM and x86 64bit binaries are register intensive compared to x86 32bit binaries — in ARM and x86 64bit architectures, there are 16 general-purpose registers, and hence registers are used more frequently than used in x86 32bit binaries which have only 8.

Considering the fact that `coreutils` binaries are relatively small with around 250 functions (on average) per binary, we conduct the second experiment using `BusyBox` (v1.21.1), which contains 2,410 functions. We observe that the obtained results are comparable with the results of `coreutils` binaries — 41.3% of the functions are best-match in `BusyBox`, while it is 35% for `coreutils` binaries on average across 16 experiments

Table 4.3: Total no. of functions selectively inlined in `coreutils` binaries compiled using `gcc` and `clang` for ARM, x86 32bit and 64bit architectures.

	ARM	C64	G64	C32	G32
Total functions	27,820	23,782	24,489	24,002	24,792
Inlined functions	1,620	776	1,002	823	1,188

shown in Fig. 4.10. This is an improvement of 27.5% over the accuracy achieved by [97] for `BusyBox`. However, for top-10 match (i.e., ranking 1-10), `coreutils` binaries achieve better results than `BusyBox` (78.4% vs. 67%). This result might be due to function size of the binary — in `BusyBox`, a function needs to be compared against on average 3,250 functions, and in `coreutils` binaries around 250 functions need to be compared.

Impact of selective inlining. As shown in Fig. 4.10, our selective inlining significantly improves the overall matching accuracy by 150% on average. In particular, when ARM architecture is used as target, selective inlining improves the total matching accuracy on average by about 400%. Table 4.3 summarizes the total number of functions inlined for each architecture. We notice that `coreutils` compiled for ARM contains more functions (27,820 across 103 binaries) than that compiled for x86 32bit (avg. 24,397) and x86 64bit (avg. 24,136). From the second row of Table 4.3, it can be seen that 61% and 82% more functions are selectively inlined by BINGO in ARM than that in x86 32bit and 64bit architectures, respectively. In ARM binaries, the larger number of compiled functions suggests that compiler inlining does not happen as frequently as it does in the other two architectures. Hence, selective inlining is required to capture the complete semantics in binary analysis, which results in the improved matching accuracy.

Summary. On `coreutils` and `BusyBox` binaries, BINGO achieves the best match (i.e., rank #1) for 35%-42% of the functions on average, for cross-architecture analysis, and selective inlining improves the result by 150% on average.

4.7.2.2 BinGo-E’s Answer to RQ1

For BINGO-E, we also match the binary functions in `coreutils` binaries compiled for one architectures (i.e., x86 32bit, x86 64bit and ARM) to that for another. We use the two compilers `gcc` (v4.8.2) and `clang` (v3.0) with the optimization level `O2` (default settings in many Linux distributions).

Figure 4.11: Cross-architecture function matching after selective inlining using `coreutils` binaries for BINGO and BINGO-E. Here, **C** and **G** represent compilers `clang` and `gcc`, respectively, and 32 and 64 denote x86 32bit and 64bit architectures.

Fig. 4.11 shows the average results after selective inlining on `coreutils` binaries for BINGO and BINGO-E. Note we only count the percentage of functions ranked 1st (i.e., best match in the case of mapping the binary functions from the same source code). In Fig. 4.11, we list 16 possible scenarios of cross-architecture code matching. For example, in the plot, the brown bar **G64-ARM** denotes the result that 62.7% of the functions (signatures) compiled by `gcc` for x86-64 are the best matches of those functions (targets) compiled by `gcc` for ARM architecture according to BINGO-E.

Previously, we observed that in Fig. 4.10 that when binary functions compiled for ARM are used as signatures, the matching results significantly degrade — only 18% of the functions are ranked 1st, based on the low-level semantic features in BINGO. The reason is that I/O values of registers in BINGO are not effective for cross-architecture matching. Hence, in BINGO-E, the newly added high-level semantic and structural features are complement to improve the matching accuracy. The results in Fig. 4.11 prove the effectiveness of these new features. For most cases, the new features improve the matching accuracy by at least 50%, compared to the results of BINGO. Especially, in the cases of code for ARM as signatures (**ARM-C32**, **ARM-C64**, **ARM-G32**, **ARM-G64**), the improvement of best match rate is above 200%. This indicates the ineffectiveness of low-level semantic features used by BINGO for matching the code for ARM to the code

for other architectures.

Currently, for BINGO-E, we get more similar results for many cases — the best match rate is within the range of 70% $\sim$ 90% for the mapping from x86-32 to x86-64 (or the other way around), while the best match rate drops to 40% $\sim$ 60% if the ARM code is used as signatures or targets. The rationale is that the BB structure of code for x86-32 or x86-64 is highly similar; even in code for ARM, the basic BB structure is also retained during the cross-architecture compilation. Hence, the structural features and high-level semantic features are the effective complements, since the CFG structures and function call information are retained during the compilation (no matter we use `gcc` or `clang`). We find that the high-level semantic features, if they are available, are even more helpful than structural features. The reason is that after selective inlining the information on function call in binary code does not change with the compilation options of architecture.

Another interesting observation is that the results for BINGO-E are much more symmetric than the results of BINGO. For example, the best match rate for BINGO-E in C64-ARM and ARM-C64 is 0.431 and 0.479, respectively. Hence, reversing signatures and targets does not affect as much as that in BINGO. In contrast, for BINGO in C64-ARM and ARM-C64 it is 0.33 and 0.162, respectively. The unsymmetrical results for BINGO are due to the step of function model generation, in which k -tracelet is sampled and generated from the CFG. As applying graph isomorphism for CFG matching, the matching of CFG g_1 to g_2 does not mean the matching of g_2 to g_1 — g_1 may be a sub-graph of g_2 and using g_2 as signatures leads to a similarity score lower than the similarity threshold. Similarly, in BINGO, low level semantic features extracted from k -tracelet (i.e., sampled partial CFG) also have this issue of unsymmetrical results.

To mitigate this, the function similarity (i.e., FDD in § 4.4.1) for 3D-CFG is defined to produce symmetrical results, according to Definition 3. In addition, the high-level semantic features (e.g., function call information) are retained in cross-architecture compilation. Hence, the results of BINGO-E are more symmetrical than that of BINGO.

In this experiment, on average, our approach significantly improves the accuracy of best match rate from 35.1% (for BINGO) to 65.6% (for BINGO-E) on `coreutils`. For cross-architecture matching on the same code base, structural features and high-level semantic features are effective, being good complement to low-level semantic features. Further, BINGO-E’s results are more symmetrical than BINGO’s.

4.7.3 Answer to RQ2: Robustness in Cross-compiler Matching

4.7.3.1 BinGo’s Answer to RQ2

Inter-compiler Matching. Table 4.4 summarizes the results obtained by BINGO for different compilers (`gcc` and `clang`) with various optimization levels (O0, O1, O2 and O3) for x86 32bit architecture. Here, each row and column of the table represents the compiler (including the level of optimization) used to compile the signature and target functions, respectively. For example, the cell denoted by the second row and fifth column (C1G0) represents the result for which signature functions are compiled using `clang` with optimization level O1 and the target functions are compiled using `gcc` with O0. From the table, we observe that BINGO is good, as 41.5% of the functions achieve rank one (on average across all the experiments for binaries compiled for x86 32bit) while this increases to 84% for top 10 matches, which is very promising. We observed similar behaviour in binaries compiled for x86 64bit architecture and due to space constraints, the results are reported in our tool webpage [133].

Interestingly, we find that regardless of the compiler type, matches between the binaries that are compiled with high code optimization levels (i.e., O2 and O3) yield better results than matches between binaries with no code optimization (i.e., O0). That is, high code optimization levels lead to the similar binary code even with different compilers, whereas without optimization the influence of the compiler in the binary is very evident. This observation is consistent with [97]. Further, within one compiler type (e.g., `gcc`), the matching accuracy achieved by optimization level O2, across all other compiler types and optimization levels, is always better or consistent with the results obtained by O3 (relevant rows are shaded in light gray in Table 4.4). This behaviour is observed in x86 64bit architecture too.

Intra-compiler Matching. In addition, we also conduct the experiments across different versions of the same compiler (`gcc` v4.8.3 vs. `gcc` v4.6) and the results (for x86 32bit architecture) are summarized in Table 4.5. As expected, the highest accuracy is achieved when the signature and target functions are of the same optimization level, except when the signature functions are compiled with optimization level O1, where on

Table 4.4: In BINGO, percentage of functions ranked #1 in inter-compiler (x86 32bit) comparison for `coreutils`. Here, **C** and **G** represent `clang` and `gcc` compilers, respectively and 0-3 represents optimization O0-O3.

	C0	C1	C2	C3	G0	G1	G2	G3
C0	-	0.402	0.373	0.372	0.371	0.4	0.451	0.332
C1	0.392	-	0.456	0.455	0.396	0.429	0.503	0.388
C2	0.345	0.426	-	0.576	0.344	0.411	0.488	0.478
C3	0.344	0.425	0.575	-	0.343	0.41	0.488	0.477
G0	0.344	0.354	0.334	0.333	-	0.374	0.398	0.302
G1	0.37	0.409	0.41	0.409	0.376	-	0.517	0.395
G2	0.412	0.462	0.474	0.473	0.428	0.514	-	0.48
G3	0.326	0.372	0.438	0.439	0.331	0.42	0.5	-

such occasions, target functions compiled with optimization level **O2** yield better accuracy. The best results obtained for each signature binary is highlighted in light gray in Table 4.5. Further, we also observe that the overall accuracy for intra-compiler analysis (42.7%) is slightly better than the inter-compiler analysis (41.5%). Finally, in intra-compiler analysis, around 76.3% functions ranked within top 5 positions, whereas it is only 68.5% for inter-compiler analysis, an 11.8% improvement. Details on the results of intra-compiler analysis for x86 64bit architecture are reported in [133].

Impact of selective inlining. We observe that selective inlining improves the average matching accuracy by around 140% for cross-compiler analysis. Many functions are inlined when the binaries are compiled with the optimization level **O0** compared to **O3**. In particular, 1,473 functions (on average across `gcc` and `clang`) are inlined in for optimization level **O0**, whereas, it is only 831 for **O3**.

Summary. For different types and versions of compiler, in most cases, BINGO attains the best match for around 42% of the functions, and top-5 match for around 72% of the functions on average.

Table 4.5: In BINGO, percentage of functions ranked #1 in intra-compiler (x86 32bit) comparison for `coreutils`. Here, **G** and **G'** represent `gcc` v4.8.3 and v4.6 compilers, respectively.

	G0	G1	G2	G3		G'0	G'1	G'2	G'3
G'0	0.43	0.36	0.36	0.27	G0	0.44	0.4	0.41	0.42
G'1	0.37	0.46	0.48	0.36	G1	0.38	0.47	0.48	0.45
G'2	0.4	0.44	0.52	0.41	G2	0.4	0.49	0.52	0.5
G'3	0.4	0.44	0.49	0.49	G3	0.32	0.39	0.43	0.52

4.7.3.2 BinGo-E's Answer to RQ2

Inter-compiler Matching. Still using `coreutils` binaries, Table 4.6 summarizes the results obtained by BINGO-E for different compilers (`gcc` and `clang`) with various optimization levels (O0, O1, O2 and O3) on `coreutils` for x86 32bit architecture. The row represents the optimization level of signature functions and the column represents that of target functions. We observe that the best results are usually achieved when O1 or O2 is adopted. The worst results (below 80%) are achieved when O3 is used, no matter for `gcc` or `clang`. Even C2-to-G1 or G2-to-C1 can achieve an accuracy above 88%. Hence, for cross-compiler matching, we find the impact of compiler differences matters less than that of the optimization levels.

In this experiment, BINGO-E (best-match rate ranging from 70.1% to 99.7% in Table 4.6) exhibits significant improvement over BINGO (best-match rate ranging from 32% to 58% in Table 4.4) for cross-compiler analysis. The improvement is attributed to the use of features of BB structure and high-level semantics, as BB structure can be retained for different compilers under the same or similar optimization levels.

Intra-compiler Matching. We also conduct intra-compiler matching for BINGO-E. Table 4.7 summarizes the searching results obtained by BINGO-E for different versions of the compiler `gcc` with various optimization levels (O0, O1, O2 and O3) for x86 32bit architecture. Here, each row and column of the table represents the compiler (including the level of optimization) used to compile the signature and target functions, respectively. For example, for the cell at row G'0 and column G0 in the left part of Table 4.7, the value 0.91 means that 91% of functions successfully match if we use the version compiled by G'0 as signature and use the function compiled by G0 as target.

Table 4.6: In BINGO-E, percentage of functions ranked #1 in inter-compiler (x86 32bit) comparison for `coreutils`. Here, **C** and **G** represent `clang` and `gcc` compilers, respectively and 0-3 represents optimization O0-O3.

	C0	C1	C2	C3	G0	G1	G2	G3
C0	-	0.865	0.791	0.785	0.799	0.840	0.836	0.701
C1	0.840	-	0.838	0.833	0.891	0.913	0.887	0.756
C2	0.814	0.889	-	0.996	0.862	0.926	0.858	0.757
C3	0.809	0.885	0.997	-	0.857	0.924	0.855	0.759
G0	0.801	0.888	0.853	0.846	-	0.884	0.856	0.754
G1	0.786	0.878	0.857	0.852	0.850	-	0.928	0.857
G2	0.806	0.882	0.814	0.808	0.848	0.963	-	0.732
G3	0.732	0.773	0.754	0.753	0.770	0.863	0.751	-

It can be observed that usually the best-match results are achieved when the signature and target functions are combined by the same optimization level (O0-O2, except O3). The reasons lie that BINGO-E adopts the structural and high-level semantic features for intra and cross compiler matching. When the same optimization level (except O3) is used for signature and target functions, the structural information is well-retained. However, when O3 is used, the structure of the binary is so over-optimized that less information is retained for structural matching. The two over-optimized program may have less structural similarity in G'3-to-G3 or G3-to-G'3 matching.

In the experiment, the results of BINGO-E (best-match rate ranging from 70% to 98% in Table 4.7) are significantly better than the results of BINGO (best-match rate ranging from 27% to 52% in Table 4.5). For the same code base with intra-compiler matching, the structural and high-level semantic features are more effective than the low-level semantic features, relying on the retained structural information.

4.7.4 Answer to RQ3: Robustness in Cross-platform (OS) Matching

4.7.4.1 BinGo's Answer to RQ3

To evaluate BINGO as a search engine for binary programs (i.e., matching against wild binaries that do not share the same code), we conduct an experiment with open-source

Table 4.7: In BINGO-E, percentage of functions ranked #1 in intra-compiler (x86 32bit) comparison for `coreutils`. Here, **G** and **G'** represent `gcc` v4.8.3 and v4.6 compilers, respectively.

	G0	G1	G2	G3		G'0	G'1	G'2	G'3
G'0	0.91	0.86	0.80	0.70	G0	0.98	0.88	0.84	0.76
G'1	0.86	0.97	0.88	0.82	G1	0.84	0.94	0.90	0.84
G'2	0.85	0.94	0.97	0.76	G2	0.80	0.92	0.97	0.81
G'3	0.79	0.88	0.89	0.75	G3	0.72	0.83	0.78	0.74

and close-source binaries. In this experiment, we choose Windows `msvcrt` as the binary (closed-source) from which the search queries are obtained, and Linux `libc` as the target binary (open-source) to be searched on.

Building Ground Truth. In conducting this experiment, we face a challenge in obtaining the ground truth. Ideally, for each function in `msvcrt`, we want to identify the corresponding semantically similar function in `libc`, so that we can faithfully evaluate BINGO. To reduce the bias in obtaining the ground truth through manual analysis, we rely on the function names and the function description provided in the official documentation, e.g., `strcpy` function in `msvcrt` is matched with `strcpy` (or its variants) from `libc`. Besides, several variants of the same functions are grouped into one family. In `libc`, there are 15 variants of `memcpy` functions are present (e.g., `memcpy_ia32`, `memcpy_sse3`, and so on). Hence, if `memcpy` from `msvcrt` is matched to any of these 15 variants from `libc`, we still consider the match is correct.

BinGo's Results. BINGO reports a total of 60 matched cases, among which 50 matched cases are true positive, according to the built ground truth between `msvcrt` and `libc`. Table 4.8 summarizes the details of the 50 matched cases: BINGO finds the best match for 11 functions (22%), top 2-5 match for 26 functions (52%), and top 6-10 match for 13 functions (26%). Since binary code matching on different code bases is much more challenging than matching on the same code base, such results have demonstrated the robustness of BINGO in identifying functions that have similar semantics but different implementations.

Summary. BINGO outperforms two available state-of-the-art tools BINDIFF and TRACY on finding the semantically similar functions in cross-OS analysis. It attains an accuracy of 51.7% and 83.3% for rankings within top 5 and 10 positions, respectively.

4.7.4.2 BinGo-E's Answer to RQ3

To evaluate the capability of BINGO-E on matching binaries from different code bases, we repeat this experiment for cross-OS matching. In this experiment, functions from Window binary `mscvrt` are used as signature functions, while functions from Linux binary `libc` are as target functions to be matched.

BinGo-E's Results. To compare with BINGO, we also analyze the same number (i.e., 60) of matched cases, which have higher similarity scores than others among results reported by BINGO-E. After examining the matching results, 56 functions are found to be true positive cases, in which one function in `mscvrt` correctly finds its counterpart in `libc` in the top-10 matching. More specifically, BINGO-E finds the best match for 32 functions (51.7%), top 2-5 match for 12 functions (21.4%), and top 6-10 match for 12 functions (21.4%).

After inspecting the results, we report the possible false positive and false negative cases. For false positive cases, some of the functions are too simple so that accidental clones are detected. For example, if both two functions have only one BB and no function calls, and are similar in the beginning checksum. Actually they have different logics after checksum, somehow, they are indistinguishable by all categories of features. For false negative cases, the `mbstowcs` function in `msvcrt` calls 6 other functions within 3 BBs. But the same function in `libc` only contains 1 BB with 2 function calls. They have similar semantics but low similarity in function calls, BB structures; emulation somehow shows different results for the two functions. In this case, BINGO-E does match the two functions as it was supposed. Thus, such semantic binary code clones are still difficult to detect.

Comparison with the State-of-the-art Tools. To compare BINGO-E with the state-of-the-art tools, we repeat the aforementioned experiment using the industry standard binary comparison tool BINDIFF⁶ and the publicly available academic tool TRACY [98]. BINDIFF (v4.1.0), somehow, is unable to correctly match any of the functions in Table 4.8 and 4.9 in `msvcrt` to their counterparts in `libc`. Its failure roots back to heavy reliance on the program structure and call-graph pattern, which is less likely to be preserved in binaries compiled from completely different source-code bases. For TRACY, only 27

⁶BinDiff: www.zynamics.com/bindiff.html

Table 4.8: BINGO’s returned rankings (top-10 matches) for 60 matched cases in `msvcrt` against `libc`

Rank	Library functions	#
1	tolower, memset, wcschr, wcsncmp, toupper, memcmp, strspn, wcsrchr, memchr, strrch,r strchr	11
2-3	wcstoul, wcstol, strtoul, fopen, strncpy, strtol, itoa, wcscmp, wcsncat, itow, longjmp, strcspn, wcsncpy, labs, strpbrk, toupper, write, memcpy, memmove, tolower	20
4-5	mbtowc, wcstombs, strcat, remove, mbstowcs, wctomb	6
6-10	rename, strstr, wcpbrk, iswctype, strtok, wscoll, strcoll, setlocale, qsort, wcspn, swprintf, wcstod, strerror	13

Table 4.9: BINGO-E’s returned rankings (top-10 matches) for 60 matched cases in `msvcrt` against `libc`

Rank	Library functions	#
1	wcscmp, sin, vswprintf, strstr, memcpy, perror, memmove, freopen, strcmp, wcschr, strcat, asin, fsetpos, strcspn, cos, strcpy, acos, qsort, wcslen, wcsncat, memcmp, wscat, malloc, swprintf, strncat, isleadbyte, strlen, fseek, wcsrchr, longjmp, strpbrk, wcsncmp	32
2-5	getc, wcpbrk, fscanf, fgetwc, strncmp, wcspn, wcsncpy, wcscspn, swscanf, strspn, tan, calloc	12
6-10	memchr, fgets, strchr, exit, strrchr, ungetwc, isprint, fopen, getchar, strerror, sscanf, wcsstr	12

functions are found to have the correct match within the top 50 positions. Among the 27 matches, 5 (8.3%) are ranked one (best-match), 23 (38.3%) are ranked within top-5 match. Notably, when matching binaries compiled for different architectures, TRACY totally fails. Due to the fact that BLEX [108] is not publicly available⁷, we cannot compare BINGO-E with dynamic analysis based function matching.

In this experiment, BINGO-E has a significant improvement on BINGO for the cross-OS analysis. In 60 analyzed cases, the false positive rate drops from 16.7% (for BINGO) to 6.7% (for BINGO-E). Among true positive cases, the rate of best-match increases from 22% (for BINGO) to 51.7% (for BINGO-E). The improvement is attributed to the adoption of high-level semantic features (e.g., system-call tags), which eliminate false positive cases due to using low-level semantic features alone.

⁷We also tried to ask for the dataset that is used to evaluate BLEX as a search engine, but we did not get any response from the authors.

4.7.5 Answer to RQ4: Applications

4.7.5.1 BinGo’s Answer to RQ4: Vulnerability Detection via Matching Open-source Software Bug to Commercial Software

As part of the on-going research, we leveraged on BINGO to perform vulnerability extrapolation [134] — given a known vulnerable code, called *vulnerability signature*, using BINGO, we try to find semantically similar vulnerable code segments in the target binary. This line of work is receiving more attention from the academic research community [97, 134]. However, there is few evidence of using such technique to hunt real-world vulnerabilities. One reason could be these tools cannot handle large complex binaries. Due to program structure agnostic function modeling, BINGO is capable of handling large binaries. Hence, we evaluate the practicality of BINGO in hunting real-world vulnerabilities.

Zero-day Vulnerability (CVE-2016-0933) Found: In vulnerability extrapolation, we discovered a zero-day vulnerability in one of 3D libraries used in Adobe PDF Reader. At the high level, we discover a network exploitable heap memory corruption vulnerability in an unspecified component of the latest version of Adobe PDF Reader. The root cause for this vulnerability is the lack of buffer size validation, which subsequently allows an unauthenticated attacker to execute arbitrary code with a low access complexity. We use a previously known vulnerability in an input size handling code segment of the same 3D module as the signature, where the signature function consists of more than 100 basic blocks. We model the known vulnerable function and all other ‘unknown’ functions in the library using BINGO, then use the known vulnerable function model as a signature, and search for semantically similar functions. BINGO returns a ‘potential’ vulnerable function (ranked #1) in the suspected 3D library. With some additional manual effort, we are able to confirm this vulnerability. Later, Adobe confirms and subsequently releases a patch for it⁸.

Matching Known Vulnerabilities: To evaluate whether BINGO is capable of finding vulnerabilities across-platform, we repeat the two experiments reported in [97]. First one is *libpurple* vulnerability (CVE-2013-6484), where this vulnerability appears in one

⁸We received 4,000USD as the bug bounty for this vulnerability **CVE-2016-0933** under *iDefense Vulnerability Contributor Program* run by *VeriSign*.

of the Windows application (**Pidgin**) and its counterpart in Mac OS X (**Adium**). In [97], it is reported that without manually crafting the vulnerability signature, matching from Windows to Mac OS X and *vice versa*, achieved the ranks #165 and #33, respectively. In BINGO, we achieve rank #1 for both cases.

The second experiment is SSL/Heartbleed bug (CVE-2014-0160), we compile the **openssl** library for Windows and Linux using **Mingw** and **gcc**, respectively (vulnerable code is shown in Fig. 4.1). The vulnerability is matched from Windows to Linux, using basic-block centric matching, similar to [97], we achieve the ranks 22 and 24 for **dtls1_process_heartbeat** and **tls1_process_heartbeat** functions, respectively. For Linux to Windows matching, we achieve rank #4 for both functions. This is due to the fact that **Mingw** significantly modifies the program structure through inlining, and hence the BB-structure is not preserved across binaries. Binary compiled by **Mingw** contains 24,923 more basic blocks compared to the Linux binary. This observation confirms that the BB centric matching fails in such conditions, where the program structure is heavily distorted. However, using function model of *k*-tracelet together with selective inlining, we are able to achieve rank #1 for both functions in Windows to Linux matching and *vice versa* with the average semantic similarity score of 0.62 and 0.59, respectively. DISCOVRE [109] fails to identify any of these vulnerable functions, as we observe that in **Mingw** version of **openssl**, library functions (e.g., **memset**, **memcpy**, and **strtol**) are inlined in most cases, leading to program structure distortion that is not handled by DISCOVRE.

4.7.5.2 BinGo-E's Answer to RQ4: New Application of Matching Binaries of Forked Projects

We also apply BINGO-E to detect the code plagiarism for the two projects that may share the same or similar code base. In this experiment, we adopt the same target projects as those used in CoP [1], namely **thttpd-2.25b** and **sthttpd-2.26.4**, where **sthttpd** is forked from **thttpd** for maintenance.

Table 4.10 shows the results of using **thttpd** as signature functions and **sthttpd** as target functions, while Table 4.11 shows the results of the matching in the other way around. In Table 4.10 and 4.11, the row refers to the compiler (including the level of

optimization) used to compile the signature functions, and the column refers to that used to compile the target functions. We observe these facts:

1. The results of these two tables are quite consistent. The reason is twofold: a) functions in `thttpd` and `sthttpd` are quite cloned and most of functions are exactly identical; b) BINGO-E's results are symmetrical as mentioned in §4.7.2.2, and hence the results are not significantly different when reversing the signature and target functions in matching.
2. With the same optimization level, functions of these two projects are perfectly matched without errors. This perfect result is attributed to the same BB structure in binary due to the same optimization level. However, CoP [1, 93] also only reports about 90% accuracy for the matching between `thttpd-2.25b` and `sthttpd-2.26.4` with the same optimization level. As CoP relies on low-level semantics (i.e., I/O values), and BINGO-E use low-level/high-level semantic and structural features, all the features are proven to be effective in matching the binary code with the same BB-structure.
3. In general, except using the same optimization level, the matchings between `gcc` G0-G2 produce the best results (the results in **Bold** font), i.e., averagely 93.8% in Table 4.10 and 93.4% in Table 4.11. For the same configuration, CoP achieves about 85% for the G0-G2 or G2-G0 matching among `thttpd-2.25b` and `sthttpd-2.26.4`. Note that no results are reported for CoP on matching binary code compiled by `gcc` G3 in [1, 93].
4. C2-to-C3 or C3-to-C2 in Table 4.10 and 4.11 all archive the accuracy of 100%. This indicates the impacts of `clang` C2 and `clang` C3 are similar. However, the impact of `gcc` G3 is different from that of `gcc` G0-G2 — `gcc` G3 usually leads to the accuracies below 80%. The rationale is that G3 would greatly change the BB-structure due to the heavy usage of compiler optimization.

The above results also exhibit how well BINGO-E can be applied to solve the typical binary clone code search problem — matching two similar projects' code under the same compilation condition (i.e., the same architecture, OS and compiler options). Note that the diagonals of Table 4.10 and 4.11 refer to the scenarios of typical binary clone

Table 4.10: In BINGO-E, percentage of functions ranked #1 in inter-compiler (x86 32bit) comparison of `thttpd` to `sthttpd`. Here, **C** and **G** represent `clang` and `gcc` compilers, respectively and 0-3 represents optimization O0-O3. **Bold** font highlights results of the configuration `gcc` O0-O2, the same as that in CoP [1].

	C0	C1	C2	C3	G0	G1	G2	G3
C0	1.0	0.820	0.761	0.757	0.922	0.830	0.864	0.711
C1	0.781	1.0	0.787	0.770	0.860	0.884	0.901	0.646
C2	0.676	0.760	1.0	1.0	0.761	0.824	0.829	0.789
C3	0.657	0.757	1.0	1.0	0.757	0.822	0.827	0.814
G0	0.961	0.883	0.803	0.800	1.0	0.895	0.876	0.737
G1	0.819	0.853	0.824	0.822	0.853	1.0	0.968	0.797
G2	0.782	0.833	0.829	0.827	0.864	0.989	1.0	0.793
G3	0.667	0.654	0.817	0.843	0.693	0.821	0.817	1.0

Table 4.11: In BINGO-E, percentage of functions ranked #1 in inter-compiler (x86 32bit) comparison of `sthttpd` to `thttpd`. Here, **C** and **G** represent `clang` and `gcc` compilers, respectively and 0-3 represents optimization O0-O3. **Bold** font highlights results of the configuration `gcc` O0-O2, the same as that in CoP [1].

	C0	C1	C2	C3	G0	G1	G2	G3
C0	1.0	0.828	0.761	0.757	0.930	0.830	0.862	0.693
C1	0.773	1.0	0.800	0.784	0.867	0.884	0.900	0.654
C2	0.676	0.747	1.0	1.0	0.774	0.824	0.829	0.789
C3	0.657	0.743	1.0	1.0	0.771	0.822	0.827	0.814
G0	0.961	0.883	0.802	0.800	1.0	0.884	0.875	0.733
G1	0.819	0.853	0.824	0.822	0.853	1.0	0.967	0.769
G2	0.800	0.846	0.829	0.827	0.854	0.978	1.0	0.780
G3	0.657	0.646	0.817	0.843	0.697	0.823	0.805	1.0

code search. For example, C0-to-C0 in Table 4.10 represents the scenario, where the signature functions are from `thttpd` and compiled with C0, and the target functions are from `sthttpd` and compiled with C0. In this study, we only use around 130 common functions between these two projects to measure the best-ranking rate in Table 4.10 and 4.11. As the diagonals in these two tables are all 1, BINGO-E achieves perfect results for the mostly-identical functions that are commonly shared between these two projects. The reason is that `sthttpd` extents `thttpd` with the dozen Gentoo patches, and rewrites the build system.

Table 4.12: Matching results between functions of **BusyBox** 1.20.0 and functions of **BusyBox** 1.21 using BINGO-E, TRACY and BINDIFF

All 2406 Functions	Rank 1	Rank 1-3	Rank 1-10
BINGO-E	95.87%	97.69%	98.38%
TRACY (k=1)	83.33%	85.95%	89.61%
BINDIFF	96.81%	N.A.	N.A.
1348 Large Functions	Rank =1	Rank 1-3	Rank 1-10
BINGO-E	99.11%	99.63%	99.85%
TRACY (k=4)	97.18%	98.00%	98.81%

In this experiment, BINGO-E demonstrates the good performance in matching binary functions of two forked projects. Especially for the matchings between gcc G0-G2, BINGO-E achieves the accuracy of 93.6% on average from `thttpd` to `sthttpd` or in the other way around. Under the same configurations, it was reported that COP achieves the accuracy of 88% on average in [1, 93].

4.7.6 Answer to RQ5: Solving the Typical Code Clone Search Problem with the Same OS, Architecture, Compiler

As discussed in Section 4.7.5.2, BINGO-E achieves the perfect matching results on the two small projects in case of the same architecture, platform and compiler configuration (the diagonals of Table 4.10 and 4.11) — the typical binary code clone detection problem. However, `thttpd` and `sthttpd` are small projects. In this experiment, we want to scale up the size of functions and compare BINGO-E with the state-of-the-art and available binary code clone detectors, namely BINDIFF and TRACY, on solving the typical binary code detection problem.

We try to run the experiment on the binaries of **Coreutils**. However, all the three tools achieve the matching results with an accuracy about 100%. Therefore, we choose to use a larger binary exactable **BusyBox** as the experiment target. To simulate the case of searching for similar functions in two projects, we use the same compiler settings (G2), same architecture (x86 32bit), and same platform (Windows) to compile two different

versions of **BusyBox**, i.e., 1.20.0 and 1.21. We choose the functions from **BusyBox** 1.20.0 as the signature functions (queries in code searching) to search for the corresponding functions in **BusyBox** 1.21.1.

Table 4.12 shows the matching results of BINGO-E, TRACY and BINDIFF on the two versions of **BusyBox**. First, we use all functions in the source (version 1.20.0) to match against all the functions in the target (version 1.21). Among all the three tools, BINDIFF has the best accuracy when examining the most similar function which be ranked at the first position. BINGO-E has a slightly lower accuracy than BINDIFF. However, its accuracies at rank 1-3 and rank 1-10 are also higher than BINDIFF rank 1. As BINDIFF only provides the best-match result, we are not able to compare rank 1-3 and rank 1-10 directly between the two tools. Both of the two tools have outperformed TRACY, whose accuracy is below 90%. Second, as TRACY claims to work better on large functions [98], we redo the experiment on only *large functions* that are with *4 or more basic blocks*. We use only 1348 functions out of the original 2406 functions. The result has shown that, for functions with a larger number of basic blocks, BINGO-E still outperforms TRACY by one or two percent. However, since both of the accuracies are around 98% to 99%, they are comparable in accuracy.

For false positives in BINDIFF, for example, the function `socket_timeout` in **BusyBox** 1.21.1 has one more instruction than the same function in **BusyBox** 1.20.0. In addition, the function is very small with only one basic block. BINDIFF has found a more similar function pair. It matches the `socket_timeout` in **BusyBox** 1.20.0 with a function named `sig_catcher_0` in **BusyBox** 1.21.1, since both of the two functions have 3 instructions. From this example, we know that when the function size is small, BINDIFF has a high chance to match different functions together. Thus, it will produce some false positives. Another type of false positives happens when the tool has finished the matching for most of the functions. The tool will match the remaining functions even if the differences between them are big. Therefore, those cases lead to the false positives. For the false positive in TRACY, for example `add_cmd_block`, the function is large with more than 10 basic blocks, and the function is slightly modified between two versions. TRACY may mistakenly match `traclet` in different functions. Although the match similarity may be low between different functions, as long as the similarity is higher than the one in the same function, TRACY will give false positive cases. In addition, TRACY may also fail when the functions are too small with the same reason as BINDIFF. Similar, BINGO-E

also faces the same issue when the function size is small and the high-level semantics (e.g., function call information) is lacking.

4.7.7 Answer to RQ6: Scalability

4.7.7.1 BinGo’s Answer to RQ6

BINGO has demonstrated its scalability through out the experiments. Its particular filtering process takes only few milliseconds for `coreutils` binaries to shortlist the candidate target functions. For example, in cross-architecture analysis, it takes 91.8 milliseconds on average to compare entire `coreutils` suite (in total, 103 binaries each containing on average 250 functions) compiled for one architecture against another, while it is reduced to 68.6 milliseconds for cross-compiler analysis. Besides, function filtering reduces the search space dramatically. After filtering, for each signature function in `coreutils` binary, on average around 21 target functions are shortlisted in cross-architecture analysis and it is further reduced to 13 functions in cross-compiler analysis. For large binaries such as `BusyBox` (around 3,250 functions with 39,179 basic-blocks), it takes on average 6.16 seconds to filter the target functions and for each signature function less than 40 target functions are shortlisted. Similarly, for SSL/Heartbleed vulnerability search in `openssl` (around 5,700 functions with more than 60K basic-blocks), filtering process takes 12.24 seconds, with only 53 functions shortlisted.

The major overhead in BINGO is the partial trace generation and semantic feature extraction operation. For example, it takes 4,469s to extract semantic features from 2,611 `libc` functions — on average, taking 1.7s to extract semantic features from a `libc` function, whereas, it takes only 1,123s to extract semantic features from 1,220 `msvcrt` functions (on average 0.9s per function). By virtue of our filtering technique, the time-consuming step of semantic feature extraction is not applied to all the functions in a binary. In practise, semantic feature extraction is a one time job that can be easily parallelized. Finally, in locating the vulnerable function in Adobe PDF Reader, it takes, in total, 88 seconds to filter and extract semantic features from the shortlisted target functions and an additional 2.7 seconds to do the semantic matching. This shows that using BINGO, it allows us to find a zero-day vulnerability in a real-world COTS binary within two minutes, provided a good signature function.

4.7.7.2 BinGo-E’s Answer to RQ6: Better Scalability

For example, in cross-architecture analysis, it takes 485.2 milliseconds on average to compare entire `coreutils` suite (in total, 103 binaries each containing on average 250 functions) compiled for one architecture against another, while it is reduced to 404.6 milliseconds for cross-compiler analysis. For experiments on the cross-OS matching, it takes 123.70s to compare entire `libc` binary against `msvcrt`. For other large binaries such as `BusyBox` (around 3,250 functions with 39,179 basic-blocks), it takes 1307.92 seconds to rank all the functions (402.4 milliseconds per function).

The major overhead in BINGO-E is using emulation to extract low-level semantic features. For example, it takes 2334.4s to extract semantic features from 2,611 `libc` functions. In the experiments we run emulation 16 times to improve the robustness. We found that the results did not change much if we run it only one time. Similarly, we were able to extract semantic features from `msvcrt` with 1,220 functions in 787.96s. For other overheads, to extract 3D-CFG features from `msvcrt`, it takes 334.73s; while it takes around 147.67s to obtain the other high-level semantic features. In practice, the feature extraction process is a one-time job that can be easily parallelized.

In this experiment, BINGO-E has significantly reduced the time for low-level semantic feature extraction. For example, the time for 1,220 `msvcrt` functions is reduced from 1,123s to 787.96s; the time for 2,611 `libc` functions is reduced from 4,469s to 2334.4s. Using the saved time, BINGO-E can extract high-level semantics and structural features to improve the accuracy.

4.7.8 Discussion

4.7.8.1 Threats to Validity

Threats to internal validity come from these aspects.

- One major internal limitation is about selective inlining — we cannot inline functions that are invoked indirectly (i.e., indirect call). However, this limitation is not particular to BINGO-E but an inherent problem in static analysis technique. Early in 2005, people have identified this problem and proposed solutions, such as VSA

(value-set analysis) [135]. However, we did not incorporate VSA into BINGO-E as it involves heavyweight program analysis, which might lead to heavy performance overheads.

- Previously, in BINGO, the constraint solving for low-level semantic features is based on IR language (i.e., REIL), which suffers from the issue of the incomplete support of floating-point (FPU) instructions. Now, IR is not needed, as the CPU emulator, UNICORN, is used. Hence, some part of internal limitation of BINGO-E comes from the limitation of UNICORN. The potential errors in handling different instruction sets may lead to errors for low-level semantic features and affect the final results.
- We adopt IDA Pro to disassemble and generate CFGs from the functions in binaries. As pointed by Meng *et al.* [127], accurate disassembling and binary analysis is not easy. The errors in CFG generation may affect the 3D-CFG features and further affect the results. In future, Dyninst can be used for accurate CFG generation.

Threats to external validity rise from these issues.

- Currently, we mainly use `coreutils` and `BusyBox` for the evaluation of cross-architecture and cross-compiler matching, `msvcrt` and `libc` for cross-OS matching. Besides, `sthttpd` and `thttpd` are used for binary code matching among forked projects. These executables are some common benchmarks that are used in other studies, e.g., `coreutils` in [97, 108], `sthttpd` and `thttpd` in [1, 93]. It is still worth exploring to what extent the existing experimental results can be true when large-size commercial software binaries are used for matching.
- For the tool comparison, we only run the available tools, BINDIFF [112] and TRACY [98], with the by-default configuration. For the comparison with DISCOVRE [109], BugSearch tool [97] and CoP [1, 93], we compare the results based on the figures that are reported in the corresponding paper of each tool. In future, when more other tools are available, we would like to run them, tune up their parameters, and conduct detailed comparison with BINGO-E.

4.7.8.2 The Focus of Our Study.

In this chapter, our goal is to combine the different categories of features and leverage emulation for acceleration. Hence, the focus of our study is not to systematically evaluate and comprehensively summarize which features are more effective for which matching scenarios or which projects. Currently, throughout the experiments, only some preliminary observations on effectiveness of different features are listed as follow:

- Low-level semantic features are effective, especially when the matching is on different code bases or the binary code obfuscation is applied [1]. However, according to our observation, false positive cases are discovered in case of 1) two binary segments are too short to contain meaningful semantics; 2) they are null check or checksum at the beginning, which leads to the match of two code segments for most give input.
- High-level semantic features capture the semantics of system/library calls, being helpful for cross-OS matching or the matching on different code bases, especially for some specific tasks of binary code search (e.g., malware detection [113]). However, for cross-OS matching, the mapping of system calls on different OSs need to be carefully handled. Besides, like Adobe Reader that implements the `malloc`, some software implements certain system/library calls, which may fail the match if such case is not considered.
- Structural features are most effective if the matching is on the same code base using similar compilers and the optimization levels [98, 110]. Thus, for code plagiarism, structural features can be effective if no binary code obfuscation is applied. The issue of structural features is usually the variance of the BB-structure. Using tracelet or 3D-CFG can be a remedy to the issue.

In future, at least a dozen of configurations (3 choices of architectures, 2 choices of OSs, 2 choices of compilers) can be set up to systematically evaluate under which configuration what features are more effective. Also, it is worth investigation that how binary or function size affects the usefulness of these features.

4.8 Related Work

Our work is relevant to the following three lines of works.

4.8.1 Code Search Using Low-level Program Semantics

Treating the program as a black box, the idea of checking input-output (I/O) values and intermediate values is applied to identify the semantic clones or behavioral clones, even without the usage or the analysis of code. This idea is first applied to detect semantic clones in source code. As early as in 2009, Jiang *et al.* [105] proposed to cluster the code fragments according to the I/O samples of random testing. In 2011, to detect semantic clones, Kim *et al.* [136] proposed the tool MECC, which models the abstract memory state and matches similar functions by comparing I/O values and guard conditions. Modelling abstract state in MECC is, to some extent, similar to that we do in BINGO for low-level semantics extraction, but MECC works on C/C++ code and our approach works on binary code.

The similar idea is applied for binary code search. In recent years, Schkufza *et al.* [106] presented a method to compare x86 loop-free snippets for testing the correctness of binary code transformation (e.g., transforming code compiled by `llvm -O0` to code compiled by `gcc -O3`). Based on I/O values and states of the machine, the equivalence is checked when the execution is complete. In addition to I/O values, intermediate values can be an alternative solution when it is difficult to identify I/O variables in binary code. Zhang *et al.* [137] proposed to use the local variables and local addresses referenced, which are extracted from dynamic execution traces, for comparing the binary code of two versions of the program. In [107], based on the assumption that some specific intermediate values are inevitable in implementing the same algorithm, Jhi *et al.* [107] proposed to use value-sequence to serve as the fingerprint for the binary function and compare value-sequence for measuring similarity. According to this assumption of considering immediate values as the “signature” of an algorithm, Zhang *et al.* [138] presented a method based on value dependency graph to detect algorithm plagiarism. In recent studies, BLEEX [108] used low-level semantic features such as the values read from or written to stack/heap and the return value of registers. In [1] and [93], Luo *et al.* adopted symbolic formulas to represent the I/O relations of blocks and applied longest

common subsequence (LCS) based fuzzy matching to measure binary code similarity, even when binary code obfuscation is present.

Among the above studies, we use the similar low-level semantic features that are used in [1, 93, 136] to model the memory states and I/O values in BINGO. Different from other studies, we extract these low-level semantics from k -traces (k -length partial trace). To overcome the drawbacks of low-level semantics in § 4.4.3.1, in BINGO-E, we incorporate some low-level semantic features proposed in BLEX [108] (§ 4.4.3.2). Note that all the above studies employ dynamic execution or constraint solving to extract the I/O values, but BINGO-E adopts emulation to speed up the extraction.

4.8.2 Code Search Using High-level Program Semantics

In the domain of malware detection and analysis, system call sequences or system call based behavior graph [139] are widely used as the signatures of malware, as they represent the unique high-level semantics of the program — the functionality the program behaves. Usually, system calls are mapped and categorized into different abstract actions that represent certain behaviors [113]. In such a way, similar malware can be detected regardless of architecture and OS differences.

System calls represent program semantics. Similarly, third party library calls can also represent high-level program semantics. BLEX [108] is presented to tolerate the differences in compilation optimization and obfuscation. As mentioned above, BLEX executes functions of the two given binaries with the same inputs and compares the output behaviors. To relax the difference of two binaries, BLEX further uses the high-level semantic features from an execution, i.e., calls to imported library function via the point and system calls made during execution. According to their report, using Library function invocation alone can only gain an accuracy of 17% in matching, while using system calls alone produces an accuracy of 38%.

Hence, for achieving a better matching accuracy, using the names of library function invocation and system calls is insufficient. Hu *et al.* [140] proposed the approach MOCKINGBIRD, which extracts function signature as a sequence of two dynamic features: Comparison Operand Pairs (COP) and System Call Attributes (SCA). COP refers to the operation result that directly determines the control flow of the execution, and SCA

refers to system call attributes that consist of names and argument values of all system calls invoked in an execution. Besides, to support cross-architecture code matching, MOCKINGBIRD addresses the diversity issue of instruction sets using VEX-IR.

In BINGO-E, we are not only using attributes of system calls and library function calls, but also incorporating opcode and op types. More than names and attributes of system calls, we also consider tags for common system calls to support cross-OS code matching. The idea of system call tags is inspired by the abstract actions that are used for malware detection [113]. Note that BINGO-E adopts no IR.

4.8.3 Code Search Using Structural Information

CFG and its derived representations are the commonly-used structural information for binary code matching. BINDIFF [112] builds CFGs of the two binaries and then adopts a heuristic to normalize and match the two CFGs. Essentially, BINDIFF resolves the NP-hard graph isomorphism problem (matching CFGs). BINHUNT [141], a tool that extends BINDIFF, is enhanced for binary diff at the two following aspects: considering matching CFGs as the Maximum Common Induced Subgraph Isomorphism problem, and applying symbolic execution and theorem proving to verify the equivalence of two basic code blocks. To address non-subgraph matching of CFGs, BINSLAYER [142] models the CFG matching problem as a bipartite graph matching problem. For these tools, compiler optimization options change the structure of CFGs and fail the graph-based matching. Besides, graph-based matching may produce unsymmetrical results.

Witnessing the weakness of graph-based matching, some researchers proposed new derived representation of binary code (e.g., graphlet [122], tracelet [98]). Saebjornsen *et al.* [104] proposed a binary code clone detection framework that leverage on normalized syntax (i.e., normalised operands) based function modelling technique. Jang *et al.* [121] propose to use n -gram models to get the complex lineage for binaries, and normalize the instruction mnemonics. Based on the n -gram features, the code search is done via checking symmetric distance. Krügel *et al.* [122] propose a graphlet-based approach to identify malware, which generates connected k -subgraphs of the CFG and apply graph-coloring to detect common subgraphs between a malware sample and a

suspicious one. Tracelet [98] is presented to capture syntax similarity of execution sequences and facilitate searching for similar functions. However, these tools relying on structural information may fail during cross-architecture and cross-OS analysis.

In BINGO-E, the structural information of CFG is represented as a new form, i.e., 3D-CFG [114]. To the best knowledge of ours, 3D-CFG is previously used for byte-code clone detection, and our study is the first attempt to apply this idea for binary code matching. Besides, to avoid the noise due to the differences in BB-structure of CFGs, before extracting low-level semantic features, we build function model of k -tracelet, and extract low-level semantic features from them. Hence, in BINGO-E, the idea of using k -tracelet [98] is adopted to mitigate the impact of differences in BB-structure.

Apart from the above studies, DISCOVRE [109] is a purely syntactic based technique and does not capture the semantics of the program. Hence, it mainly performs cross-architecture matching tasks on the binaries that share the same source code but compiled for different architectures. However, if the two binaries compiled for different architectures do not share the same source code, DISCOVRE cannot handle them. On the other hand, Multi-MH [97] considers the program semantics in a limited fashion. That is, Multi-MH's semantics are limited to individual functions (i.e., consider each function in isolation) and it doesn't consider how each function interacts with each other. The problem in considering each function in isolation is that it cannot tackle the function inlining/outlining problems that arise do to various optimization techniques commonly used by modern compilers. Hence, both tools are not designed for binary code matching tasks of cross-architecture, cross-platform and cross-compiler configuration.

4.8.4 Security Application of Binary Code Search

In cyber security domain, binary code search tool is a powerful weapon for securing binary executables (e.g., malware detection and vulnerability discovery). In many scenarios, static analysis tools are preferred for binary code auditing. Dynamic analysis faces challenges from two aspects: the difficulty in setting up the execution environment, and the scalability issue that prevents large-scale detection.

Pinpointed by Zaddach *et al.* [143], these dynamic approaches are far from practical application onto highly-customized hardware like mobile or embedded devices. Thus it

is difficult for these approaches to conduct cross-architecture bug detection. To address this issue, Pewny *et al.* [97] and DISCOVRE [109] propose a static analysis technique with the aim to detect vulnerabilities inside multiple versions of the same program compiled for different architectures. Thus, their approach is good for finding clones of the same program due to architecture or compilation differences, not suitable for finding semantic binary clones among different applications. [144] aims to detect infected virus from executables via a static CFG matching approach. In addition, [145] and [146] also adopt CFG in the recovery of the information of compilers and even authors.

In this study, we make as signature the vulnerable binary functions from open source projects, and search them in the closed-source commercial executable (e.g., Adobe PDF Reader). The same or similar vulnerability can be discovered, if an open source software bug is forked into the commercial solution.

4.9 Conclusion

In this work, we present the scalable solution of binary code search framework, BINGO-E, which aims to search similar binary code regardless of the differences in architecture, OS and compilation options. Rather than only relying on low-level semantic features in BINGO, we further incorporate high-level semantic features and structural features in BINGO-E. To speed up low-level feature extraction, we adopt emulation in BINGO-E. The promising experimental results live up to the expectation towards effective and efficient cross-architecture, cross-OS and cross-compiler binary code search. Further, BINGO-E has outperformed the state-of-the-art tools like TRACY and BINDIFF, and also significantly improved the previous version of our tool (i.e., BINGO).

As advance in code clone search can facilitate many tasks on program binary analysis, many possible applications of BINGO-E need to be explored. For example, in security application, we have discovered a zero-day vulnerability in Adobe PDF Reader. In future, our ambition is to apply BINGO-E on matching open-source software bugs to the commercial software's binaries, in order to reveal more unknown vulnerabilities due to the forking of common open-source projects and spreading the bugs inside them. Due

to the popularity of IoT (Internet of Things), the cross-OS, cross-architecture and cross-compiler binary code clone search is expected to aid in tasks of matching vulnerable binary code between Windows and IoT.

5

Identifying Potential Vulnerable Functions via Program Analysis

5.1 Introduction

Vulnerabilities have become the major threat to cyber-security. Security experts often use fuzzing (e.g., [147–150]), symbolic execution (e.g., [151–154]) and manual auditing to hunt vulnerabilities. As only a few vulnerabilities are scattered across a large amount of code, vulnerability hunting is a very challenging task that requires intensive knowledge and skills [155]. Therefore, a large amount of time and effort is wasted analyzing those non-vulnerable code. In that sense, identifying potentially vulnerable code in a code base can guide vulnerability hunting and assessment in the promising direction (i.e., putting more time and effort to where it is most needed), and thus improve the effectiveness of

vulnerability hunting and assessment. In this thesis, we are focusing on the potential vulnerability detection at source level for the open source projects, which can help to find the binary vulnerability.

Although attack surface identification has been attracting great attention, some problems still remain. For example, metric-based techniques are mostly designed for one particular application. Thus, they might not work on other applications because machine learning may over-fit to some noise features. Moreover, while an empirical connection between vulnerabilities and bugs exist, the connection is considerably weak due to the differences between vulnerabilities and bugs [184]. Hence, the research on bug prediction cannot directly translate to attack surface identification. Unfortunately, the existing metric-based techniques use the similar metrics as those in bug prediction, and thus fail to investigate the characteristics of vulnerabilities.

In this thesis, we propose an tool, named PVHUNT, to identify potentially vulnerable functions in C/C++ applications. PVHUNT is designed to be *generic* to work for all kinds of applications, *lightweight* to support the analysis of real-world applications, and *extensible* with domain-specific data to improve the accuracy. PVHUNT is proposed as a pre-step for vulnerability assessment, i.e., to guide security experts during the further in-depth analysis by narrowing down the space of potentially vulnerable functions. Different from metric-based and pattern-based techniques, PVHUNT does not require any prior knowledge about known vulnerabilities.

Our framework works in two steps by combining two sets of systematically derived program metrics, i.e., complexity metrics and vulnerability metrics. Complexity metrics capture the complexity of a function in four dimensions: the function itself, the loop structures in the function, the control structures in the function, and the dependency of the function. Vulnerability metrics reflect the characteristics of memory corruption vulnerabilities in three dimensions: the constants, arrays, and pointers in a function. We focus on memory corruption vulnerabilities as they are the most common ones.

We implemented the proposed framework in Python to obtain complexity and vulnerability metrics for each project. We evaluated the effectiveness, scalability and applications of our framework with ten different types of real-life projects. The results have demonstrated that our framework can cover 66% of vulnerabilities by identifying 20%

of functions as potentially vulnerable. Based on the identified vulnerable functions in the current stable release of PHP, we discovered six zero-day vulnerabilities.

In summary, our work makes the following contributions.

- We developed a lightweight vulnerable function identification tool, which incorporates two sets of program metrics to identify potentially vulnerable functions.
- We conducted comprehensive experiments on ten real-world applications to demonstrate the effectiveness, scalability and applications of our framework.
- We found several unknown vulnerabilities in current stable release of PHP based on the identified vulnerable functions.

5.2 Methodology

In this section, we first present the overview of our framework, and then elaborate each step of our framework in detail.

5.2.1 Overview

Fig. 5.1 presents the overview of PVHUNT, which is designed to be lightweight. The input is the source code of a target C/C++ application. It performs first function binning and then function ranking, and returns potentially vulnerable functions for vulnerability assessment.

In the first step (Section 5.2.2), we use complexity metrics to group all functions in the target program into a set of bins. The complexity metrics basically reflect the complexity of a function in various dimensions: the function itself (e.g., Cyclomatic complexity), the loop structures in the function (e.g., the number of nested loops), the control structures in the function (e.g., the number of control structures), and the dependency of the function (e.g., fan-out). Each bin has a different level of complexity, which is designed to identify vulnerabilities at all levels of complexity. In this way, we can avoid missing vulnerable functions with low-complexity.

Figure 5.1: An Overview of the Proposed Framework

In the second step (Section 5.2.3), we use vulnerability metrics to rank the functions in each bin, and identify the top functions in each bin as potentially vulnerable. The vulnerability metrics capture the characteristics of memory corruption vulnerabilities in three dimensions: the constants in a function (e.g., the number of constants), the arrays in a function (e.g., the number of array variables), and the pointers in a function (e.g., the number of pointer arithmetic). Notice that we focus on memory corruption vulnerabilities in this chapter as they have been the most common and dangerous vulnerabilities. Through incorporating such vulnerability metrics, we can avoid identifying buggy functions as vulnerable functions.

5.2.2 Function Binning

Different vulnerabilities often have different levels of complexity. To identify vulnerabilities at all levels of complexity, in the first step, we categorize all functions in the target application into a set of bins based on complexity metrics. As a result, each bin represents a different level of complexity. Instead of directly predicting potential vulnerability using complexity metrics as is done by the existing metric-based techniques, we leverage complexity metrics to split the target application into bins where the second step (Section 5.2.3) plays the prediction role via ranking. Such a binning-and-ranking approach is designed to avoid missing low-complexity vulnerable functions.

Complexity Metrics. We derive the complexity metrics of a function based on two views: the internal structure of the function and the external relationship with other functions. Internally, a function often has loop and control structures, which are the main sources of structural complexity. Externally, a function can be called by or call other functions, which reflects the interaction complexity. Based on this understanding,

we introduce the complexity of a function with respect to four dimensions, as shown in Table 5.1. The first three dimensions are related to the internal structure of a function.

- Function metric (C1) that captures the overall Cyclomatic complexity [185] of a function, i.e., the number of linearly independent paths through a function. A higher value of C1 indicates that the function is more likely to be difficult to test.
- Loop structure metrics (C2–C4) that reflect the complexity of loops, which are challenging in program analysis [186] and hinder vulnerability analysis, i.e., the number of loops, the number of nested loops, and the maximum nesting level of loops. The higher these metrics, the more difficult to analyze the function.
- Control structure metrics (C5–C9) that capture the complexity of control structures (i.e., *if*, *switch*, *for* and *while*), i.e., the number of control structures, the number of nested control structures, the maximum nesting level of control structures, and the maximum number of control structures that are control- or data-dependent. The higher these metrics, the more difficult to reach the deeper part of the function during vulnerability hunting.
- Dependency metrics (C10–C11) that characterize the dependency relationship of a function with other functions, i.e., the number of functions calling the function (fan-in) and the number of functions called by the function (fan-out). The more dependent with other functions, the more difficult to track the interaction.

Binning Strategy. Given the metrics for all functions in the target program, we compute a *complexity score* for each function by summing up the metric values of this function. After that we group the functions that have the same score into the one bin. Note that we do not give the bin a range, that is to group the functions whose scores fall into the same range into one bin. This is because it is hard to determine the suitable granularity of the range. This kind of simple method makes our solution a lightweight, and also give good results as shown in the experimental study (see Section 5.3.3).

Roles in Vulnerability Assessment. These metrics can also enable security experts to decide what program analysis techniques can be applied during the assessment of those identified vulnerable functions, and how to prioritize their analysis. For example, if a vulnerable function has a relatively low score of complexity metrics, symbolic execution

Table 5.1: The Complexity Metrics of a Function

Dimension	ID	Metric Description
CD1: Function	C1	Cyclomatic complexity
CD2: Loop	C2	# of loops
Structures	C3	# of nested loops
	C4	Maximum nesting level of loops
CD3: Control Structures	C5	# of control structures (if, switch, for, and while)
	C6	# of nested control structures
	C7	Maximum nesting level of control structures
	C8	Maximum of control-dependent control structures
	C9	Maximum of data-dependent control structures
CD4: Dependency	C10	# of functions calling the function (Fan-In)
	C11	# of functions called by the function (Fan-Out)

can be a good choice to confirm if there is a vulnerability in the function; and high priority should be first given to such low-complexity functions because the potential vulnerabilities in such functions will be most likely first discovered by attackers.

5.2.3 Function Ranking

Different vulnerability types usually have different causes. Here we focus on memory corruption vulnerabilities that have been the most common and dangerous methods for attackers to threaten software security. To precisely identify potential vulnerable functions, we derive a new set of vulnerability metrics based on the possible evidence of memory corruption bugs in the second step. And then we rank the functions and identify the top ones in each bin as potentially vulnerable functions based on the vulnerability metrics. Our solution is different from the metric-based techniques, which do not use any vulnerability related metrics. We use vulnerability metrics to avoid recognizing general software bugs as vulnerabilities.

Vulnerability Metrics. In general, memory corruption vulnerabilities are caused by incorrectly using programming features of memory management and pointer arithmetic [187]. In particular, pointers are commonly used to access and modify memory; and dereferencing a null or out-of-bounds pointer triggers the vulnerability. The array

Table 5.2: The Vulnerability Metrics of a Function

Dimension	ID	Metric Description
VD1: Constants	V1	# of constants involved in loop predicates
	V2	# of constants involved in control predicates
VD2: Arrays	V3	# of array variables
	V4	# of array variables involved in loop predicates
	V5	# of array variables involved in control predicates
VD3: Pointers	V6	# of pointer variables
	V7	# of pointer variables involved in loop predicates
	V8	# of pointer variables involved in control predicates
	V9	# of pointer arithmetic
	V10	# of variables involved in pointer arithmetic
	V11	Maximum pointer arithmetic a variable is involved in
	V12	Average pointer arithmetic a variable is involved in

in C/C++ is manipulated by means of a pointer to the first element. Hence, array variables are almost identical to pointers. Moreover, constants are widely used for “bounds checking” in the predicates (i.e., conditional expressions) of loop and control structures. If developers fail to check for correct bound values, array indexing and pointer arithmetic may attempt to read from or write to a location outside of the space allocated for them and trigger the vulnerability. Based on these observations of memory corruption vulnerabilities, we introduce the vulnerability metrics of a function with respect to three dimensions, as reported in Table 5.2.

- Constant metrics (V1–V2) that capture the used constants in predicates, i.e., the number of constants used in the predicates of loop and control structures. A higher value of these metrics indicates a higher chance of erroneously checking the bounds.
- Array metrics (V3–V5) that reflect the usage of array variables, i.e., the number of array variables, and the number of array variables used in loop and control predicates. A higher value of these metrics indicates that it is more likely to dereference null or out-of-bounds array variables.
- Pointer metrics (V6–V12) that capture the manipulation of pointers, i.e., the number of pointer variables, the number of pointer variables used in the predicates

of loop and control structures, the number of pointer arithmetic, the number of variables used in pointer arithmetic, and the maximum and average number of pointer arithmetic a variable is involved in. The higher these metrics, the higher chance to dereference null or out-of-bounds pointers.

Notice that there usually exists a proportional relation between complexity and vulnerability metrics because the more complex the (loop and control) structures of a function, the higher chance the constants, arrays and pointers are involved in the structures. Hence, instead of directly ranking the functions with complexity and/or vulnerability metrics, we propose a binning-and-ranking approach to avoid missing those low-complexity but vulnerable functions, as will be evidenced in Section 5.3.2.

5.3 Evaluation

5.3.1 Evaluation Setup

To demonstrate the effectiveness, scalability and applications of our framework, we conducted experiments on ten real-life projects.

Target Applications. We used ten real-life open-source projects that represent a diversity of applications. `glibc` is the GNU C Library that provides the core libraries for the GNU system as well as GNU/Linux systems. `ffmpeg` is the leading multimedia framework. `QEMU` is a generic machine emulator and virtualizer. `Asterisk` is a free framework for building custom communications solutions and innovative products. `libxml2` is the XML C parser and toolkit developed for the Gnome project. `libxslt` is the XSLT C library for the GNOME project. `OpenSSL` is a robust and full-featured toolkit for the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols. `Wireshark` is a network traffic analyzer for Unix and Unix-like operating systems. `Linux` is a monolithic Unix-like computer operating system kernel. `PHP` is a popular general-purpose scripting language that is especially suited to web development.

The details of each target application are reported in Table 5.3. The first column gives the project version, the second column reports the release date, and the third column lists the total number of functions in each project. The last three columns report the

Table 5.3: Details of the Target Applications

Project	Release Date	Func. (#)	Vul. Func. (#)	CVE (#)	Excluded CVE (#)
glibc 2.22	2015-08	9299	11	4	0
ffmpeg 3.1.3	2016-08	19240	19	10	0
QEMU 2.5.0	2015-12	31880	5	3	0
Asterisk 12.0.0	2013-12	18906	24	12	0
libxml2 2.9.4	2016-05	5364	25	12	4
libxslt 1.1.28	2012-11	677	5	3	0
OpenSSL 1.0.1t	2016-05	6712	21	10	1
Wireshark 2.2.0	2016-09	32789	36	22	1
Linux 4.8.11	2016-11	440761	11	9	2
PHP 5.6.30	2017-01	10013	0	0	0

number of vulnerable functions, CVEs and excluded CVEs, collected as ground truth via manual analysis (see below). Here, we chose the recent versions of the projects that had large number of CVEs. The number of functions ranges from 677 for `libxslt` to 440761 for `Linux`, which is diverse enough to show the scalability of our framework.

Ground Truth. To obtain the ground truth for evaluating the effectiveness of our framework, we first manually identified the known vulnerabilities that were discovered before January 2017 in the ten projects from two vulnerability database websites: CVE Details [189] and National Vulnerability Database [190]. Then, we manually analyzed each vulnerability to determine the functions that are patched to fix the vulnerability and mark them as vulnerable. The results are reported in the fourth and fifth columns of Table 5.3. The total effort was around 1.5 man-months.

In particular, during our manual analysis, we excluded some CVEs, as shown in the last column of Table 5.3 because (i) there is no public detail about the fix that can directly identify the affected vulnerable functions by source code analysis (i.e., CVE-2016-4614, CVE-2016-4615, CVE-2016-4616 and CVE-2016-4619 for `libxml2 2.9.4`); (ii) the fix does not involve direct code change (i.e., CVE-2016-7958 for `Wireshark 2.2.0` and CVE-2016-2183 for `OpenSSL 1.0.1t`); and (iii) there is no fix due to certain specific reasons (i.e., CVE-2010-5321 and CVE-2015-2877 for `Linux 4.8.11`). We also excluded some vulnerable functions that are actually affected by some CVEs, because Joern misses

Figure 5.2: Vulnerability Score vs. Complexity Score for Non-Vulnerable Functions (in Blue) and Vulnerable Functions (in Red)

such functions due to its own limitations (e.g., the function `__strftime_internal` in `strftime.1.c` for `glibc 2.22`, affected by CVE-2015-8776).

5.3.2 Rationality of Binning-and-Ranking

To evaluate whether our binning-and-ranking approach is reasonable, we first computed the complexity score and vulnerability score, as introduced in Section 5.2.2 and Section 5.2.3, for each non-vulnerable and vulnerable functions in all the projects except for `PHP` which does not have any publicly known vulnerable functions (as shown in Table 5.3). Then we plotted in Fig. 5.2 the relationship between complexity score (i.e., x -axis) and vulnerability score (i.e., y -axis) using logarithmic scale, where vulnerable and non-vulnerable functions were respectively highlighted in red and blue.

We can see from Fig. 5.2 that all projects share the similar patterns; vulnerable functions are scattered across non-vulnerable functions with respect to complexity score and vulnerability score; and there is an approximately proportional relation between complexity

score and vulnerability score for vulnerable functions. Therefore, if we directly ranked the functions based on complexity metrics and/or vulnerability metrics, we would always favor those functions with high complexity score and high vulnerability score and miss those low-complexity but vulnerable functions (e.g., the lower left red circles in Fig. 5.2a, 5.2b, 5.2c, 5.2d, 5.2g and 5.2h). Instead, by first binning the functions according to complexity score and then ranking the functions in each bin according to vulnerability score, our framework can effectively identify the potentially vulnerable functions at all levels of complexity (see details in Section 5.3.3). Moreover, Fig. 5.2 also visually indicates the severe imbalance between non-vulnerable and vulnerable functions (see the third and fourth columns of Table 5.3), which indicates machine learning will over-fit and cannot be applicable to attack surface identification effectively.

Summary: Our binning-and-ranking approach is reasonable for predicting vulnerable functions at all levels of complexity.

5.3.3 Effectiveness of Binning-and-Ranking

To evaluate the effectiveness of our binning-and-ranking approach, we ran PVHUNT on all the projects except for PHP; and we also varied the parameter k (see Section 5.2.3) to analyze its impact on the effectiveness. Here we used the percentage of functions (i.e., *Iden. Func.*) that are identified by PVHUNT as potentially vulnerable, and the percentage of vulnerable functions (i.e., *Cov. Vul. Func.*) and CVEs (i.e., *Cov. CVE*) that are covered by those identified potentially vulnerable functions as the three indicators of the effectiveness of our framework. Here a CVE is said to be covered if one of its affected vulnerable functions is covered. These three indicators are used throughout the evaluation section.

The results are shown in Fig. 5.3, where the x -axis represents the parameter k and the y -axis represents the percentage value of the indicators. The legends are only shown in Fig. 5.3a and omitted in others for clarity. In general, as k increases, the three indicators also increase. For an appropriate k , our binning-and-ranking approach can achieve a high value for *Cov. Vul. Func.* and *Cov. CVE* and a low value for *Iden. Func.*. This means by identifying a small part of functions as vulnerable, we cover a large portion of vulnerable functions and CVEs, which can narrow down the analysis space for security experts for effective vulnerability assessment.

Figure 5.3: Percentage of Functions (*Iden. Func.*) that are Identified as Potentially Vulnerable, and Percentage of Vulnerable Functions (*Cov. Vul. Func.*) and CVEs (*Cov. CVE*) that are Covered by Those Identified Potentially Vulnerable Functions as k Increases

Table 5.4: The Appropriate k for Each Project

Project	Appr. k	Bins (#)	Iden. Func. (%)	Cov. Vul. Func. (%)	Cov. CVE (%)
glibc 2.22	4	431	12.2	54.5	75.0
ffmpeg 3.1.3	12	368	12.6	47.4	70.0
QEMU 2.5.0	8	330	3.5	20.0	33.3
Asterisk 12.0.0	20	410	22.3	41.7	75.0
libxml2 2.9.4	3	242	10.5	56.0	91.7
libxslt 1.1.28	1	128	20.5	60.0	100
OpenSSL 1.0.1t	4	264	11.0	71.4	80.0
Wireshark 2.2.0	11	466	7.8	55.6	81.8
Linux 4.8.11	7	1039	0.8	18.2	33.3

Detailedly, we chose the appropriate k for each project to be the first k after which *Cov. CVE* is not changed during a large range of k . The results are shown in Table 5.4.

The second column reports the appropriate k for each project, and the third to sixth columns list the corresponding number of bins generated in the binning step and the three indicators. We omit the results for PHP because it has no CVE. We can observe that for small-scale projects (e.g., `glibc`, `libxml2`, `libxslt` and `OpenSSL`), k can be set to a small value (e.g., 4), and we identify 13% functions as vulnerable and cover 60% vulnerable functions and 87% CVEs on average; and for large-scale projects, k can be set to a large value (e.g., 12), and we identify 10% functions as vulnerable and cover 37% vulnerable functions and 59% CVEs on average. These results demonstrate that our binning-and-ranking approach is effective in narrowing down the space of potentially vulnerable functions for security experts. Notice that our binning-and-ranking approach is less effective on `QEMU` and `Linux` because most vulnerable functions are not in the top of each bin, as visually shown in Fig. 5.2. The possible reasons are that (i) `QEMU` and `Linux` have diverse functionalities but are frequently released (e.g., every one or two weeks), and thus security testing is only conducted on a small set of modules (e.g., four of the five vulnerable functions in `QEMU` is in `dev-network.c`), which makes many potentially vulnerable modules not properly tested; and (ii) both `QEMU` and `Linux` are critical systems with a long history, and many vulnerable modules have been already patched.

Furthermore, we also report how many potentially vulnerable functions we need to identify to cover certain percentage of CVEs in Table 5.5, and how many CVEs are covered when we identify certain percentage of functions as vulnerable in Table 5.6. In particular, to cover 50% (or 75%) of CVEs, k can be set to a small value except for `ffmpeg`, `QEMU` and `Linux`, and on average 16% (or 27%) of functions are identified as vulnerable. To achieve full coverage of CVEs, 61% of functions are identified as vulnerable, which becomes less useful for security experts. On the other hand, when we identify 5 %, 10%, 20% and 30% of functions as vulnerable, we can cover 45%, 52%, 66% and 73% of CVEs, which has greatly narrowed down the space of vulnerable functions for security experts' vulnerability assessment.

False Negative Analysis. By closely looking into the vulnerable functions that PVHUNT needs to identify a large part of functions as vulnerable to cover, we find that logical vulnerabilities cause all the false negatives. Logical vulnerabilities, in general, involve functions with extremely low vulnerability score, such that PVHUNT needs to identify almost all functions as vulnerable to cover them. For example, the function

Table 5.5: Covered CVEs vs. Identified Vulnerable Functions

Project		Cov. CVEs (%)			
		25%	50%	75%	100%
glibc 2.22	k	1	1	4	28
	Iden. Func. (%)	4.4	4.4	12.2	58.9
ffmpeg 3.1.3	k	2	10	39	60
	Iden. Func. (%)	3.3	10.9	50.4	87.4
QEMU 2.5.0	k	8	52	53	53
	Iden. Func. (%)	3.5	43.8	46.3	48.8
Asterisk 12.0.0	k	2	10	20	43
	Iden. Func. (%)	3.7	11.7	22.3	82.5
libxml2 2.9.4	k	1	1	3	12
	Iden. Func. (%)	4.7	4.7	10.5	40.2
libxslt 1.1.28	k	1	1	1	1
	Iden. Func. (%)	20.5	20.5	20.5	20.5
OpenSSL 1.0.1t	k	1	1	4	19
	Iden. Func. (%)	4.0	4.0	11.0	47.8
Wireshark 2.2.0	k	1	3	10	48
	Iden. Func. (%)	1.5	3.1	7.3	87.9
Linux 4.8.11	k	7	93	106	112
	Iden. Func. (%)	0.8	39.3	66.3	76.9

`ldss_init_protocol` in **Wireshark** contains a logical vulnerability, but it has only one line of code and thus its complexity score is 1 and its vulnerability score is 0. In addition, we also observed that some logical vulnerabilities have a very low vulnerability score and a reasonable complexity score, which again made them difficult to detect by PVHUNT. For example, one of the logical vulnerabilities in **Asterisk** (CVE-2014-4046) allowed an unauthorized user to execute arbitrary shell commands. The fix to this permission escalation vulnerability is quite straightforward, where the user requires either a `call` or `system` permission to execute the shell commands and the patch only modified 3 lines of code in `load_module` function in the `MixMonitor` module. PVHUNT misses the function because it has the vulnerability score of 0 and a complexity score of 10. Domain knowledge may potentially help to boost the scores to reduce the false negatives.

We did not conduct false positive analysis because some vulnerabilities are secretly patched by the maintainers [191] and it is impossible to confirm whether they are indeed

Table 5.6: Identified Vulnerable Functions vs. Covered CVEs

Project		Iden. Func. (%)			
		5%	10%	20%	30%
glibc 2.22	k	1	3	8	16
	Cov. CVE (%)	50.0	50.0	75.0	75.0
ffmpeg 3.1.3	k	3	8	19	27
	Cov. CVE (%)	40.0	40.0	70.0	70.0
QEMU 2.5.0	k	13	26	36	45
	Cov. CVE (%)	33.3	33.3	33.3	33.3
Asterisk 12.0.0	k	3	8	18	23
	Cov. CVE (%)	33.3	41.7	66.7	75.0
libxml2 2.9.4	k	1	2	6	9
	Cov. CVE (%)	50.0	66.7	91.7	91.7
libxslt 1.1.28	k	-	-	-	1
	Cov. CVE (%)	-	-	-	100
OpenSSL 1.0.1t	k	1	3	8	13
	Cov. CVE (%)	50.0	70.0	80.0	90.0
Wireshark 2.2.0	k	6	15	26	32
	Cov. CVE (%)	68.2	81.8	81.8	86.4
Linux 4.8.11	k	59	71	81	88
	Cov. CVE (%)	33.3	33.3	33.3	33.3

false positives.

Summary: Our binning-and-ranking approach is effective, i.e., identifying 20% of functions as vulnerable while covering 66% of CVEs on average. Such a small portion of functions can be very useful for security experts, as will be shown in our application of PVHUNT to fuzzing in Section 5.3.7.

5.3.4 Sensitivity of the Metrics

To evaluate the sensitivity of the complexity and vulnerability metrics to our framework, we first removed one of the dimensions of the complexity and vulnerability metrics from PVHUNT, and then ran PVHUNT on all the projects except for PHP. For space limitations, we show the results of complexity metrics in Fig. 5.4, and report the similar

Figure 5.4: Sensitivity Analysis Results of the Complexity Metrics

results of vulnerability metrics at our website [188]. The x -axis and y -axis in Fig. 5.4 represent the *Iden. Func.* and *Cov. CVE* indicator.

We can see from Fig. 5.4 that if the control structure metrics (i.e., CD3) are removed, the effectiveness of our framework is reduced on most of the projects; i.e., less CVEs are covered when the same percentage of functions are identified as vulnerable. This indicates that control structure metrics contribute significantly to our framework in most of the projects. If we remove one of the other dimensions, the effectiveness is quite competitive to ours; i.e., *Cov. CVE* is high in some ranges of *Iden. Func.* for some projects, but low in others. These results indicate that it is difficult or even impossible to derive an optimal model for the metric combination that can work well for all ranges of *Iden. Func.* for all projects. The results for vulnerability metrics are similar. Hence, we design a generic but not optimal model that treats each metric equally. Notice that when we identify 25%–30% of functions as vulnerable (to cover $\geq 70\%$ of CVEs), our framework is the best for all projects except for `libxml2` and `Linux`.

Summary: Control structure metrics and pointer metrics significantly contribute to PVHUNT; and it is difficult to derive an optimal metric model that works for all projects, which motivates our generic model without sacrificing much in effectiveness.

5.3.5 Predictability of the Metrics

To evaluate whether the potentially vulnerable functions identified in an older version of an application will become truly vulnerable in its later versions, we ran PVHUNT on an older version of a project, then chose four later versions and collected the ground truth (i.e., the number of CVEs and vulnerable functions), and finally analyzed how many vulnerable functions in each version were already predicted as vulnerable in the older version. As the ground truth collection was time-consuming, we only used three of the ten projects.

Table 5.7 reports the details of the older version and four later versions of the selected three projects. Fig. 5.5 shows the analysis results, where x -axis represents the percentage of functions identified as potentially vulnerable in the older version, and y -axis represents the percentage of vulnerable functions in each later version that were predicted as vulnerable in the older version. By identifying 20% of functions as vulnerable in the older version, we can predict 33%, 64% and 65% of the vulnerable functions in later versions. These results indicate that although the identified potentially vulnerable functions are not confirmed in the current version, there is a high chance that they will become truly vulnerable in future versions.

Summary: PVHUNT can predict the vulnerable functions in later versions based on the metrics data of the current version.

Table 5.7: Details of the Older Version and Four Later Versions of the Three Projects

Older Version	Four Later Versions											
	Ver.	Vul. Func. #	CVEs	Ver.	Vul. Func. #	CVEs	Ver.	Vul. Func. #	CVEs	Ver.	Vul. Func. #	CVEs
ffmpeg 0.5	1.0.1	21	21	1.2.6	10	10	2.8.1	8	8	3.1.3	19	10
libxml2 2.7.8	2.9.1	15	3	2.9.2	19	13	2.9.3	7	4	2.9.4	25	12
OpenSSL 1.0.1a	1.0.1e	128	68	1.0.1h	117	57	1.0.1j	96	45	1.0.1t	21	10

Figure 5.5: Predictability Analysis Results of the Three Projects

5.3.6 Scalability of Our Framework

To evaluate the scalability of our framework, we collected the performance of extracting complexity and vulnerability metrics using Joern [183] and the performance of identifying potentially vulnerable functions by PVHUNT. The results are reported in Table 5.8 for all the projects. The time for building the code property graph and querying the graph to obtain metric values depends on the number of functions in each project. For small-scale projects, it respectively takes 4 and 24 minutes to build and query the graph; and it takes hours for large-scale projects (i.e., **Wireshark** and **Linux**). It takes less than 1.1 minutes to identify potentially vulnerable functions for all projects except for **Linux**. These results demonstrate that our framework scales well for large-size projects like **Linux**.

Summary: Our framework scales well and can be applied to large-scale applications like **Linux**.

5.3.7 Applications of Our Framework

Here we discuss several application scenarios of the identified potentially vulnerable functions to vulnerability assessment.

Target Identification for Fuzzing. The identified potentially vulnerable functions can be used as the targets for fuzzing, helping security experts focus on interesting functions. Experimentally, we ran PVHUNT on **PHP 5.6.30**, and used the identified 500 potentially vulnerable functions as the targets for fuzzing. Notice that **PHP** is used by more than 80% of all the websites, and 5.6.30 is the current stable version. This means **PHP** is well-tested by its users, developers, and security researchers, and it is difficult to find vulnerabilities.

Using our domain knowledge about PHP, we first excluded the functions in those well-fuzzed modules (e.g., SQLite, phar and gd), and then focused on the functions in the modules (e.g., mbstring and Zend) that are related to file system and network data as they are often reachable through entry points. Finally, we developed specific testing drivers to reach and fuzz those functions, and discovered six vulnerabilities in PHP 5.6.30, which are also reproducible in another stable branch PHP 7.1.5. Table ?? reports the details of the six vulnerabilities, including the affected module, the type, and if the vulnerability is reproducible in 32-bit and 64-bit Ubuntu OS. We have analyzed these vulnerabilities, produced patches, and reported them. Six CVE numbers are assigned, but are omitted for anonymity. Moreover, it took only 1.5 hours to discover the first vulnerability in PHP, and only one week to discover all the six vulnerabilities.

Seed Selection for Fuzzing. Fuzzers often keep interesting test inputs (i.e., seeds) for further fuzzing. By giving high priority to the seeds, which reach the functions identified by PVHUNT, during seed selection, the penetration power of fuzzers can be improved. Experimentally, the windows port of the AFL fuzzer [192] took 8 days to find 750 new program execution paths at deterministic stage. With our optimized seed selection, it took only 1 day to find 800 new paths.

Other Applications. First, selective instrumentation for fuzzing can be achieved by only instrumenting those functions identified by PVHUNT. Compared to whole instrumentation in existing fuzzers, it can generate more focused feedbacks to direct fuzzers to more interesting functions, especially for large-scale applications. Second, targeted seed generation for fuzzing can be realized by leveraging the functionalities of the functions identified by PVHUNT during seed generation [193]. Hence, targeted seeds can be generated to reach interesting functions. Third, as a major method to find vulnerabilities, manual auditing can be greatly simplified w.r.t. the effectiveness and efficiency. Instead of examining all the functions [194], security experts can limit their review to only those functions selected by PVHUNT as potentially vulnerable.

Summary: As a pre-step of vulnerability assessment, PVHUNT can be applied to various vulnerability assessment tasks. Our preliminary application on fuzzing has shown its great potential.

Table 5.8: Performance of Our Framework

Project	Com. & Vul. Metrics		Identify Vul. Func.
	Build Graph	Query	
glibc 2.22	3.7 m	15.5 m	0.1 m
ffmpeg 3.1.3	6.6 m	49.9 m	0.4 m
QEMU 2.5.0	4.3 m	40.4 m	0.3 m
Asterisk 12.0.0	4.9 m	33.9 m	1.1 m
libxml2 2.9.4	1.2 m	10.1 m	0.02 m
libxslt 1.1.28	0.4 m	1.7 m	0.01 m
OpenSSL 1.0.1t	1.5 m	10.9 m	0.03 m
Wireshark 2.2.0	1.2 h	2.0 h	0.4 m
Linux 4.8.11	1.3 h	10.0 h	22.2 m
PHP 5.6.30	8.9 m	27.5 m	0.2 m

5.4 Conclusions

We develop a lightweight approach which is named PVHUNT, to identify potential vulnerable functions via program analysis. Experimental results on ten real-world programs show the effectiveness and scalability of PVHUNT for these applications.

6

Conclusions and Future Research

For scalable malware detection and triaging, we proposed a bounded feature space behavior modeling technique (called, eBOFM) which, when combined with appropriate machine learning and clustering techniques, can lead to effective and scalable malware detection and clustering (or triaging). The malware features extracted using eBOFM are very expressive in nature and provide valuable information to security analysts to build malware protection mechanisms. Most importantly, the eBOFM feature space is bounded by an upper limit N , a constant. Using large-scale experiments, we show that, with eBOFM, computation time and memory usage are vastly lower than in existing detection and clustering techniques, while still improving their detection and clustering accuracy.

In the future, we plan to build an online malware detection framework based on eBOFM models, where eBOFM features are extracted on-the-fly and are compared against the learnt malware behavior models. As demonstrated in this study, the malware feature

reduction along with the dramatic reduction in time and memory requirements achieved by eBOFM models are very encouraging. Therefore, we strongly believe that these advantages will enable us to perform online malware detection in a scalable and efficient manner

To learn security patching at level, we proposed a patch analysis framework SPAIN to automatically learn the security patch patterns and vulnerability patterns, and identify them from the program binary executables. It has built the bridge from the binary diffing to the automatic patch understanding. The experiments have shown that SPAIN can correctly locate more than half of the vulnerability patches in the binaries and find n-day vulnerabilities in major commercial software. SPAIN can be useful in vulnerability and patch understanding, similar bug hunting, binary code auditing, and, eventually, the program security enhancement. In the future, we plan to extend the framework to generate more detail and precise report of each patch and vulnerability through program slicing and symbolic execution, as well as to make it capable of analyzing binaries with other instruction sets, like ARM. Also, we would like to set up binary vulnerability databases with the help of the tool.

To perform binary matching, we present a scalable binary code search framework, BINGO, which aims to search similar binary code regardless of the differences in architecture, OS and compilation options. Rather than only relying on low-level semantic features in BINGO, we further incorporate high-level semantic features and structural features in BINGO-E. To speed up low-level feature extraction, we adopt emulation in BINGO. The promising experimental results live up to the expectation towards effective and efficient cross-architecture, cross- OS and cross-compiler binary code search. Further, BINGO-E has outperformed the state-of-the-art tools like TRACY and BIN-DIFF. In security application, we also discovered a zero-day vulnerability in Adobe PDF Reader. In future, we are planning to apply BINGO-E on matching open-source software bugs to the commercial software binaries, in order to reveal more unknown vulnerabilities due to the forking of common open-source software.

To detect potential vulnerabilities in software systems, we develop a lightweight approach which is named PVHUNT, to identify potential vulnerable functions via program analysis. Experimental results on ten real-world programs show the effectiveness and scalability of PVHUNT for these applications.

Bibliography

- [1] 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*. ACM, 2014, pp. 389–400.
- [2] A. Moser, C. Kruegel, and E. Kirda, “Limits of static analysis for malware detection,” in *Twenty-Third Annual Computer Security Applications Conference (ACSAC 2007)*, 2007, pp. 421–430.
- [3] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X.-y. Zhou, and X. Wang, “Effective and efficient malware detection at the end host.” in *USENIX security symposium*, vol. 4, no. 1, 2009, pp. 351–366.
- [4] A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, “Access-miner: using system-centric models for malware protection,” in *Proceedings of the 17th ACM conference on Computer and communications security*. ACM, 2010, pp. 399–412.
- [5] D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, “A quantitative study of accuracy in system call-based malware detection,” in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 2012, pp. 122–132.
- [6] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. X. Song, “Dynamic spyware analysis,” in *ATC’07 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference*, 2007, p. 18.

- [7] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan, “Synthesizing near-optimal malware specifications from suspicious behaviors,” in *2010 IEEE Symposium on Security and Privacy*, 2010, pp. 45–60.
- [8] T. Kwon and Z. Su, “Modeling high-level behavior patterns for precise similarity analysis of software,” in *Data Mining (ICDM), 2011 IEEE 11th International Conference on*. IEEE, 2011, pp. 1134–1139.
- [9] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, “Scalable, behavior-based malware clustering,” in *NDSS*, vol. 9. Citeseer, 2009, pp. 8–11.
- [10] M. Chandramohan, H. B. K. Tan, L. C. Briand, L. K. Shar, and B. M. Padmanabhuni, “A scalable approach for malware detection through bounded feature space behavior modeling,” in *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 2013, pp. 312–322.
- [11] K. Rieck, P. Trinius, C. Willems, and T. Holz, “Automatic analysis of malware behavior using machine learning,” *Journal of Computer Security*, vol. 19, no. 4, pp. 639–668, 2011.
- [12] T. Lee, J. Mody, Y. Lin, A. Marinescu, and A. Polyakov, “Application behavioral classification,” Jun. 14 2007, uS Patent App. 11/608,625.
- [13] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario, “Automated classification and analysis of internet malware,” in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2007, pp. 178–197.
- [14] Y. Xue, Z. Xu, M. Chandramohan, and Y. Liu, “Accurate and scalable cross-architecture cross-os binary code search with emulation,” *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.
- [15] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, and F. Song, “Spain: security patch analysis for binaries towards understanding the pain and pills,” in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 462–472.
- [16] S. Das, Y. Liu, W. Zhang, and M. Chandramohan, “Semantics-based online malware detection: Towards efficient real-time protection against malware,” *IEEE*

- Transactions on Information Forensics and Security*, vol. 11, no. 2, pp. 289–302, 2016.
- [17] M. Chandramohan, H. B. K. Tan, and L. K. Shar, “Scalable malware clustering through coarse-grained behavior modeling,” in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, p. 27.
- [18] M. Chandramohan and H. B. K. Tan, “Detection of mobile malware in the wild,” *IEEE Computer*, vol. 45, no. 9, pp. 65–71, 2012.
- [19] J. Jang, D. Brumley, and S. Venkataraman, “Bitshred: feature hashing malware for scalable triage and semantic analysis,” in *Proceedings of the 18th ACM conference on Computer and communications security*. ACM, 2011, pp. 309–320.
- [20] Y. Xue, J. Wang, Y. Liu, H. Xiao, J. Sun, and M. Chandramohan, “Detection and classification of malicious javascript via attack behavior modelling,” in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, 2015, pp. 48–59.
- [21] A. Walenstein, M. Venable, M. Hayes, C. Thompson, and A. Lakhotia, “Exploiting similarity between variants to defeat malware,” in *Proc. BlackHat DC Conf*, 2007.
- [22] M. Braverman, J. Williams, and Z. Mador, “Microsoft security intelligence report january-june 2006,” 2006.
- [23] 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*. Springer, 2008, pp. 108–125.
- [24] X. Hu, S. Bhatkar, K. Griffin, and K. G. Shin, “Mutantx-s: scalable malware clustering based on static features,” in *USENIX ATC’13 Proceedings of the 2013 USENIX conference on Annual Technical Conference*, 2013, pp. 187–198.
- [25] I. You and K. Yim, “Malware obfuscation techniques: A brief survey,” in *Broadband, Wireless Computing, Communication and Applications (BWCCA), 2010 International Conference on*. IEEE, 2010, pp. 297–300.

- [26] B. Stock, B. Livshits, and B. Zorn, “Kizzle: A signature compiler for exploit kits,” 2017.
- [27] M. E. Russinovich, D. A. Solomon, and A. Ionescu, *Windows internals*. Pearson Education, 2012.
- [28] M. Sikorski and A. Honig, *Practical malware analysis: the hands-on guide to dissecting malicious software*. no starch press, 2012.
- [29] F. Nicolas, L. O. Murchu, and E. Chien, “W32. stuxnet dossier,” *White paper, Symantec Corp., Security Response*, vol. 5, no. 6, p. 29, 2011.
- [30] K. G. Shin and X. Hu, “Large-scale malware analysis, detection, and signature generation,” 2011.
- [31] M. Christodorescu, S. Jha, and C. Kruegel, “Mining specifications of malicious behavior,” in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2007, pp. 5–14.
- [32] S. Ray and R. H. Turi, “Determination of number of clusters in k-means clustering and application in colour image segmentation,” in *Proceedings of the 4th international conference on advances in pattern recognition and digital techniques*. Calcutta, India, 1999, pp. 137–143.
- [33] C. Willems, T. Holz, and F. Freiling, “Toward automated dynamic malware analysis using cwsandbox,” *IEEE Security & Privacy*, vol. 5, no. 2, 2007.
- [34] I. Steinwart and A. Christmann, *Support Vector Machines*, 2008.
- [35] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, “Andromaly: a behavioral malware detection framework for android devices,” *intelligent information systems*, vol. 38, no. 1, pp. 161–190, 2012.
- [36] X. Hu and K. G. Shin, “Duet: integration of dynamic and static analyses for malware clustering with cluster ensembles,” in *Proceedings of the 29th annual computer security applications conference*. ACM, 2013, pp. 79–88.

- [37] T. Wang, X. Hu, S. Meng, and R. Sailer, “Reconciling malware labeling discrepancy via consensus learning,” in *2014 IEEE 30th International Conference on Data Engineering Workshops*, 2014, pp. 84–89.
- [38] D. Sculley, “Web-scale k-means clustering,” in *Proceedings of the 19th international conference on World wide web*, 2010, pp. 1177–1178.
- [39] C. Wressnegger, G. Schwenk, D. Arp, and K. Rieck, “A close look on n -grams in intrusion detection: anomaly detection vs. classification,” in *Proceedings of the 2013 ACM workshop on Artificial intelligence and security*, 2013, pp. 67–76.
- [40] R. Perdisci *et al.*, “Vamo: towards a fully automated malware clustering validity analysis,” in *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 2012, pp. 329–338.
- [41] D. J. Sheskin, *Handbook of parametric and nonparametric statistical procedures*. crc Press, 2003.
- [42] Y. Qiao, J. He, Y. Yang, and L. Ji, “Analyzing malware by abstracting the frequent itemsets in api call sequences,” in *2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications*, 2013, pp. 265–270.
- [43] P. Indyk and R. Motwani, “Approximate nearest neighbors: towards removing the curse of dimensionality,” in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, 1998, pp. 604–613.
- [44] L. Martignoni, E. Stinson, M. Fredrikson, S. Jha, and J. C. Mitchell, “A layered architecture for detecting malicious behaviors,” in *RAID ’08 Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, 2008, pp. 78–97.
- [45] E. Stinson and J. C. Mitchell, “Characterizing bots’ remote control behavior,” in *DIMVA ’07 Proceedings of the 4th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2007, pp. 89–108.
- [46] V. Ganesh, T. Leek, and M. Rinard, “Taint-based directed whitebox fuzzing,” in *ICSE*, 2009, pp. 474–484.

- [47] S. K. Cha, M. Woo, and D. Brumley, “Program-adaptive mutational fuzzing,” in *SECP*, 2015, pp. 725–741.
- [48] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, “Driller: Augmenting fuzzing through selective symbolic execution,” in *NDSS*, 2016.
- [49] C. Hu, Z. Li, J. Ma, T. Guo, and Z. Shi, “File parsing vulnerability detection with symbolic execution,” in *TASE*, 2012, pp. 135–142.
- [50] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw, “Its4: a static vulnerability scanner for c and c++ code,” in *ACSAC*, 2000, pp. 257–267.
- [51] J. Heffley and P. Meunier, “Can source code auditing software identify common vulnerabilities and be used to evaluate software security?” in *HICSS*, 2004, pp. 90277–.
- [52] J. Wang, B. Chen, L. Wei, and Y. Liu, “Skyfire: Data-driven seed generation for fuzzing,” in *SECP*, 2017.
- [53] F. Long and M. Rinard, “Automatic patch generation by learning correct code,” in *POPL*, 2016, pp. 298–312.
- [54] D. Kim, J. Nam, J. Song, and S. Kim, “Automatic patch generation learned from human-written patches,” in *ICSE*, 2013, pp. 802–811.
- [55] S. Mechtaev, J. Yi, and A. Roychoudhury, “Directfix: Looking for simple program repairs,” in *ICSE*, 2015, pp. 448–458.
- [56] Y. Tian, J. Lawall, and D. Lo, “Identifying linux bug fixing patches,” in *ICSE*, 2012, pp. 386–396.
- [57] C. S. Corley, N. A. Kraft, L. H. Etzkorn, and S. K. Lukins, “Recovering traceability links between source code and fixed bugs via patch analysis,” in *TEFSE*, 2011, pp. 31–37.
- [58] D. Gao, M. K. Reiter, and D. Song, “Binhunt: Automatically finding semantic differences in binary programs,” in *ICICS*, 2008, pp. 238–255.

- [59] T. Schreck, S. Berger, and J. Göbel, “Bissam: Automatic vulnerability identification of office documents,” in *DIMVA*, 2013, pp. 204–213.
- [60] F. Araujo, K. W. Hamlen, S. Biedermann, and S. Katzenbeisser, “From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation,” in *CCS*, 2014, pp. 942–953.
- [61] E. C. R. Shin, D. Song, and R. Moazzezi, “Recognizing functions in binaries with neural networks,” in *USENIX Security*, 2015, pp. 611–626.
- [62] Hex-Rayd, “IDA Pro,” <http://www.datarescue.com/idabase>, Last 2016: Aug. 2016.
- [63] H. Flake, “Structural comparison of executable objects,” in *DIMVA*, 2004, pp. 161–173.
- [64] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos, “An in-depth analysis of disassembly on full-scale x86/x64 binaries,” in *USENIX Security*, 2016, pp. 583–600.
- [65] K. Chen, P. Liu, and Y. Zhang, “Achieving accuracy and scalability simultaneously in detecting application clones on android markets,” in *ICSE*, 2014, pp. 175–186.
- [66] 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 *FSE*, 2014, pp. 389–400.
- [67] Intel, “Intel 64 and ia-32 architectures software developer’s manual,” <http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html>, Last 2016: Aug. 2016.
- [68] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Cho, and H. B. K. Tan, “Bingo: cross-architecture cross-os binary search,” in *FSE*, 2016, pp. 678–689.
- [69] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, “Cross-architecture bug search in binary executables,” in *S&P*, 2015, pp. 709–724.
- [70] M. Cova, V. Felmetzger, G. Banks, and G. Vigna, “Static detection of vulnerabilities in x86 executables,” in *ACSAC*, 2006, pp. 269–278.

- [71] J. Pewny, F. Schuster, L. Bernhard, T. Holz, and C. Rossow, “Leveraging semantic signatures for bug search in binary programs,” in *ACSAC*, 2014, pp. 406–415.
- [72] J. Park, M. Kim, B. Ray, and D. H. Bae, “An empirical study of supplementary bug fixes,” in *MSR*, 2012, pp. 40–49.
- [73] SPAIN, <http://pat.scse.ntu.edu.sg/spain/>.
- [74] A. Nistor, P.-C. Chang, C. Radoi, and S. Lu, “Caramel: Detecting and fixing performance problems that have non-intrusive fixes,” in *ICSE*, 2015, pp. 902–912.
- [75] F. Yamaguchi, M. Lottmann, and K. Rieck, “Generalized vulnerability extrapolation using abstract syntax trees,” in *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 2012, pp. 359–368.
- [76] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, *BAP: A Binary Analysis Platform*, 2011, pp. 463–469.
- [77] S. Letian, F. Jianming, C. Jing, and P. Guojun, “PVDF: An automatic Patch-based Vulnerability Description and Fuzzing method,” in *CSC*, 2014, pp. 1–8.
- [78] M. Soto, F. Thung, C.-P. Wong, C. Le Goues, and D. Lo, “A deeper look into bug fixes: Patterns, replacements, deletions, and additions,” in *MSR*, 2016, pp. 512–515.
- [79] M. Kim and D. Notkin, “Discovering and representing systematic code changes,” in *ICSE*, 2009, pp. 309–319.
- [80] Y. Zhang, X. Sun, Y. Deng, L. Cheng, S. Zeng, Y. Fu, and D. Feng, “Improving accuracy of static integer overflow detection in binary,” in *RAID*, 2015, pp. 247–269.
- [81] T. Wang, T. Wei, Z. Lin, and W. Zou, “Intscope: Automatically detecting integer overflow vulnerability in X86 binary using symbolic execution,” in *NDSS*, 2009.
- [82] Y. Shoshitaishvili, R. Wang, C. Hauser, C. Kruegel, and G. Vigna, “Firmallice - automatic detection of authentication bypass vulnerabilities in binary firmware,” in *NDSS*, 2015.

- [83] D. Brumley, P. Poosankam, D. X. Song, and J. Zheng, “Automatic patch-based exploit generation is possible: Techniques and implications,” in *SE&P*, 2008, pp. 143–157.
- [84] J. Feist, L. Mounier, and M. Potet, “Statically detecting use after free on binary code,” *J. Computer Virology and Hacking Techniques*, vol. 10, no. 3, pp. 211–217, 2014.
- [85] S. Cheng, J. Yang, J. Wang, J. Wang, and F. Jiang, “Loongchecker: Practical summary-based semi-simulation to detect vulnerability in binary code,” in *Trust-Com*, 2011, pp. 150–159.
- [86] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, “Automatic inference of search patterns for taint-style vulnerabilities,” in *SE&P*, 2015, pp. 797–812.
- [87] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, “A first step towards automated detection of buffer overrun vulnerabilities,” in *NDSS*, 2000.
- [88] T. Avgerinos, S. K. Cha, B. L. T. Hao, and D. Brumley, “AEG: automatic exploit generation,” in *NDSS*, 2011.
- [89] V. D’Silva, M. Payer, and D. X. Song, “The correctness-security gap in compiler optimization,” in *SPW*, 2015, pp. 73–87.
- [90] “The gpl-violations.org project,” <http://gpl-violations.org/>, accessed: 2016-09-22.
- [91] “Vmware lawsuit - software freedom conservancy,” <https://sfconservancy.org/news/2015/mar/05/vmware-lawsuit/>.
- [92] A. Hemel, K. T. Kalleberg, R. Vermaas, and E. Dolstra, “Finding software license violations through binary code clone detection,” in *Proceedings of the 8th International Working Conference on Mining Software Repositories, MSR 2011 (Co-located with ICSE), Waikiki, Honolulu, HI, USA, May 21-28, 2011, Proceedings*, 2011, pp. 63–72. [Online]. Available: <http://doi.acm.org/10.1145/1985441.1985453>
- [93] 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*, vol. PP, no. 99, pp. 1–1, 2017.

- [94] J. Caballero, N. M. Johnson, S. McCamant, and D. Song, "Binary code extraction and interface identification for security applications," DTIC Document, Tech. Rep., 2009.
- [95] D. Kim, W. N. Sumner, X. Zhang, D. Xu, and H. Agrawal, "Reuse-oriented reverse engineering of functional components from x86 binaries," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 1128–1139.
- [96] J. Ming, D. Xu, and D. Wu, "Memoized semantics-based binary diffing with application to malware lineage inference," in *ICT Systems Security and Privacy Protection*. Springer, 2015, pp. 416–430.
- [97] J. Powny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz, "Cross-architecture bug search in binary executables," in *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, 2015, pp. 709–724. [Online]. Available: <http://dx.doi.org/10.1109/SP.2015.49>
- [98] Y. David and E. Yahav, "Tracelet-based code search in executables," in *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, 2014, p. 37. [Online]. Available: <http://doi.acm.org/10.1145/2594291.2594343>
- [99] S. K. an Seunghoon Woo, H. Lee, and H. Oh, "Vuddy: A scalable approach for vulnerable code clone discovery," in *Proceedings of the 38th IEEE Symposium on Security and Privacy*, 2017, p. to appear.
- [100] T. Kamiya, S. Kusumoto, and K. Inoue, "Ccfinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Software Eng.*, vol. 28, no. 7, pp. 654–670, 2002. [Online]. Available: <http://doi.ieeecomputersociety.org/10.1109/TSE.2002.1019480>
- [101] L. Jiang, G. Mishherghi, Z. Su, and S. Glondu, "DECKARD: scalable and accurate tree-based detection of code clones," in *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, 2007, pp. 96–105. [Online]. Available: <http://dx.doi.org/10.1109/ICSE.2007.30>
- [102] P. P. F. Chan and C. S. Collberg, "A method to evaluate CFG comparison algorithms," in *2014 14th International Conference on Quality Software*,

- Allen, TX, USA, October 2-3, 2014*, 2014, pp. 95–104. [Online]. Available: <http://dx.doi.org/10.1109/QSIC.2014.28>
- [103] M. Gabel, L. Jiang, and Z. Su, “Scalable detection of semantic clones,” in *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, 2008, pp. 321–330. [Online]. Available: <http://doi.acm.org/10.1145/1368088.1368132>
- [104] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, “Detecting code clones in binary executables,” in *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 2009, pp. 117–128.
- [105] L. Jiang and Z. Su, “Automatic mining of functionally equivalent code fragments via random testing,” in *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009*, 2009, pp. 81–92. [Online]. Available: <http://doi.acm.org/10.1145/1572272.1572283>
- [106] E. Schkufza, R. Sharma, and A. Aiken, “Stochastic superoptimization,” in *Architectural Support for Programming Languages and Operating Systems, ASPLOS ’13, Houston, TX, USA - March 16 - 20, 2013*, 2013, pp. 305–316. [Online]. Available: <http://doi.acm.org/10.1145/2451116.2451150>
- [107] Y. Jhi, X. Wang, X. Jia, S. Zhu, P. Liu, and D. Wu, “Value-based program characterization and its application to software plagiarism detection,” in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu, HI, USA, May 21-28, 2011*, 2011, pp. 756–765. [Online]. Available: <http://doi.acm.org/10.1145/1985793.1985899>
- [108] M. Egele, M. Woo, P. Chapman, and D. Brumley, “Blanket execution: Dynamic similarity testing for program binaries and components,” in *USENIX Security Symposium*, 2014.
- [109] E. Sebastian, Y. Khaled, and G.-P. Elmar, “discovre: Efficient cross-architecture identification of bugs in binary code,” in *In Proceedings of the 23rd Network and Distributed System Security Symposium*. NDSS, 2016.

- [110] 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 2016, Santa Barbara, CA, USA, June 13-17, 2016*, 2016, pp. 266–280. [Online]. Available: <http://doi.acm.org/10.1145/2908080.2908126>
- [111] M. Chandramohan, Y. Xue, Z. Xu, Y. Liu, C. Y. Choo, and H. B. K. Tan, “Bingo: Cross-architecture cross-os binary search,” in *Proceedings of 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE’16)*, 2016, p. to appear.
- [112] H. Flake, “Structural comparison of executable objects,” in *Detection of Intrusions and Malware & Vulnerability Assessment, GI SIG SIDAR Workshop, DIMVA 2004, Dortmund, Germany, July 6-7, 2004, Proceedings*, 2004, pp. 161–173. [Online]. Available: <http://subs.emis.de/LNI/Proceedings/Proceedings46/article2970.html>
- [113] D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda, “A quantitative study of accuracy in system call-based malware detection,” in *International Symposium on Software Testing and Analysis, ISSSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, 2012, pp. 122–132. [Online]. Available: <http://doi.acm.org/10.1145/2338965.2336768>
- [114] K. Chen, P. Liu, and Y. Zhang, “Achieving accuracy and scalability simultaneously in detecting application clones on android markets,” in *36th International Conference on Software Engineering, ICSE ’14, Hyderabad, India - May 31 - June 07, 2014*, 2014, pp. 175–186. [Online]. Available: <http://doi.acm.org/10.1145/2568225.2568286>
- [115] “Unicorn: the ultimate cpu emulator,” <http://www.unicorn-engine.org/>, accessed: 2016-09-22.
- [116] “Interactive disassembler professional,” <https://www.hex-rays.com/products/ida/>, accessed: 2017-11-02.
- [117] “Dyninst for program binary analysis and instrumentation,” <https://github.com/dyninst>, accessed: 2017-11-02.

- [118] “Angr: A python framework for analyzing binaries,” <https://angr.io/>, accessed: 2017-11-02.
- [119] K. Chen, P. Liu, and Y. Zhang, “Achieving accuracy and scalability simultaneously in detecting application clones on android markets,” in *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*, 2014, pp. 175–186.
- [120] “Qemu: A generic and open source machine emulator and virtualizer,” <https://www.qemu.org/>, accessed: 2017-11-02.
- [121] J. Jang, M. Woo, and D. Brumley, “Towards automatic software lineage inference,” in *Proceedings of the 22th USENIX Security Symposium, Washington, DC, USA, August 14-16, 2013*, 2013, pp. 81–96. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity13/technical-sessions/papers/jang>
- [122] C. Krügel, E. Kirda, D. Mutz, W. K. Robertson, and G. Vigna, “Polymorphic worm detection using structural information of executables,” in *Recent Advances in Intrusion Detection, 8th International Symposium, RAID 2005, Seattle, WA, USA, September 7-9, 2005, Revised Papers*, 2005, pp. 207–226. [Online]. Available: http://dx.doi.org/10.1007/11663812_11
- [123] M. Wang, H. Yin, A. V. Bhaskar, P. Su, and D. Feng, “Binary code continent: Finer-grained control flow integrity for stripped binaries,” in *Proceedings of the 31st Annual Computer Security Applications Conference*. ACM, 2015, pp. 331–340.
- [124] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W.-M. W. Hwu, “Profile-guided automatic inline expansion for c programs,” *Software: Practice and Experience*, vol. 22, no. 5, pp. 349–369, 1992.
- [125] 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*. ACM, 2010, pp. 927–938.
- [126] G. Meng, Y. Xue, Z. Xu, Y. Liu, J. Zhang, and A. Narayanan, “Semantic modelling of android malware for effective malware comprehension, detection, and classification,” in *Proceedings of the 25th International Symposium on Software Testing*

- and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, 2016, pp. 306–317.
- [127] X. Meng and B. P. Miller, “Binary code is not easy,” in *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, 2016, pp. 24–35. [Online]. Available: <http://doi.acm.org/10.1145/2931037.2931047>
- [128] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, “MAPO: mining and recommending API usage patterns,” in *ECOOP 2009 - Object-Oriented Programming, 23rd European Conference, Genoa, Italy, July 6-10, 2009. Proceedings*, 2009, pp. 318–343.
- [129] Y. Kawakoya, M. Iwamura, E. Shioji, and T. Hariu, “API chaser: Anti-analysis resistant malware analyzer,” in *Research in Attacks, Intrusions, and Defenses - 16th International Symposium, RAID 2013, Rodney Bay, St. Lucia, October 23-25, 2013. Proceedings*, 2013, pp. 123–143.
- [130] L. De Moura and N. Bjørner, “Z3: An efficient smt solver,” in *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.
- [131] R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.
- [132] T. Dullien and S. Porst, “Reil: A platform-independent intermediate representation of disassembled code for static code analysis,” *Proceeding of CanSecWest*, 2009.
- [133] “BinGo: Cross-Architecture Cross-OS Binary Search,” <https://sites.google.com/site/bingofse2016/>, 2016, [Online; accessed 11-March-2016].
- [134] 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*. ACM, 2014, pp. 406–415.
- [135] G. Balakrishnan, T. Reps, D. Melski, and T. Teitelbaum, “Wysinwyx: What you see is not what you execute,” in *Verified software: theories, tools, experiments*. Springer, 2005, pp. 202–213.

- [136] H. Kim, Y. Jung, S. Kim, and K. Yi, “Mecc: memory comparison-based clone detector,” in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, 2011, pp. 301–310. [Online]. Available: <http://doi.acm.org/10.1145/1985793.1985835>
- [137] X. Zhang and R. Gupta, “Matching execution histories of program versions,” in *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, 2005, pp. 197–206. [Online]. Available: <http://doi.acm.org/10.1145/1081706.1081738>
- [138] F. Zhang, Y. Jhi, D. Wu, P. Liu, and S. Zhu, “A first step towards algorithm plagiarism detection,” in *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, 2012, pp. 111–121. [Online]. Available: <http://doi.acm.org/10.1145/2338965.2336767>
- [139] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang, “Effective and efficient malware detection at the end host,” in *18th USENIX Security Symposium, Montreal, Canada, August 10-14, 2009, Proceedings*, 2009, pp. 351–366. [Online]. Available: http://www.usenix.org/events/sec09/tech/full_papers/kolbitsch.pdf
- [140] Y. Hu, Y. Zhang, J. Li, and D. Gu, “Cross-architecture binary semantics understanding via similar code comparison,” in *IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering, SANER 2016, Suita, Osaka, Japan, March 14-18, 2016 - Volume 1*, 2016, pp. 57–67. [Online]. Available: <http://dx.doi.org/10.1109/SANER.2016.50>
- [141] D. Gao, M. K. Reiter, and D. X. Song, “Binhunt: Automatically finding semantic differences in binary programs,” in *Information and Communications Security, 10th International Conference, ICICS 2008, Birmingham, UK, October 20-22, 2008, Proceedings*, 2008, pp. 238–255. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-88625-9_16
- [142] 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 2013, PPREW@POPL 2013*,

- January 26, 2013, Rome, Italy, 2013*, pp. 4:1–4:10. [Online]. Available: <http://doi.acm.org/10.1145/2430553.2430557>
- [143] J. Zaddach, L. Bruno, A. Francillon, and D. Balzarotti, “AVATAR: A framework to support dynamic security analysis of embedded systems’ firmwares,” in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23–26, 2014*, 2014. [Online]. Available: <http://www.internetsociety.org/doc/avatar-framework-support-dynamic-security-analysis-embedded-systems%E2%80%99firmwares>
- [144] D. Bruschi, L. Martignoni, and M. Monga, “Detecting self-mutating malware using control-flow graph matching,” in *Detection of Intrusions and Malware & Vulnerability Assessment, Third International Conference, DIMVA 2006, Berlin, Germany, July 13–14, 2006, Proceedings*, 2006, pp. 129–143. [Online]. Available: http://dx.doi.org/10.1007/11790754_8
- [145] N. E. Rosenblum, X. Zhu, and B. P. Miller, “Who wrote this code? identifying the authors of program binaries,” in *Computer Security - ESORICS 2011 - 16th European Symposium on Research in Computer Security, Leuven, Belgium, September 12–14, 2011. Proceedings*, 2011, pp. 172–189. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-23822-2_10
- [146] 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’10, Toronto, Ontario, Canada, June 5–6, 2010*, 2010, pp. 21–28. [Online]. Available: <http://doi.acm.org/10.1145/1806672.1806678>
- [147] T. Wang, T. Wei, G. Gu, and W. Zou, “Taintscope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection,” in *SP*, 2010, pp. 497–512.
- [148] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, “Scheduling black-box mutational fuzzing,” in *CCS*, 2013, pp. 511–522.
- [149] M. Böhme, V.-T. Pham, and A. Roychoudhury, “Coverage-based greybox fuzzing as markov chain,” in *CCS*, 2016, pp. 1032–1043.

- [150] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, “Vuzzer: Application-aware evolutionary fuzzing,” in *NDSS*, 2017.
- [151] P. Godefroid, M. Y. Levin, and D. A. Molnar, “Automated whitebox fuzz testing,” in *NDSS*, 2008.
- [152] D. Babić, L. Martignoni, S. McCamant, and D. Song, “Statically-directed dynamic automated test generation,” in *ISSTA*, 2011, pp. 12–22.
- [153] S. K. Cha, M. Woo, and D. Brumley, “Program-adaptive mutational fuzzing,” in *SP*, 2015, pp. 725–741.
- [154] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, “Driller: Augmenting fuzzing through selective symbolic execution,” in *NDSS*, 2016.
- [155] T. Zimmermann, N. Nagappan, and L. Williams, “Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista,” in *ICST*, 2010, pp. 421–428.
- [156] T. Gyimothy, R. Ferenc, and I. Siket, “Empirical validation of object-oriented metrics on open source software for fault prediction,” *IEEE Trans. Softw. Eng.*, vol. 31, no. 10, pp. 897–910, 2005.
- [157] C. Catal and B. Diri, “A systematic review of software fault prediction studies,” *Expert Syst. Appl.*, vol. 36, no. 4, pp. 7346–7354, 2009.
- [158] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, “A systematic literature review on fault prediction performance in software engineering,” *IEEE Trans. Softw. Eng.*, vol. 38, no. 6, pp. 1276–1304, 2012.
- [159] D. Radjenovic, M. Hericko, R. Torkar, and A. Zivkovic, “Software fault prediction metrics: A systematic literature review,” *Information and Software Technology*, vol. 55, no. 8, pp. 1397–1418, 2013.
- [160] R. Malhotra, “A systematic review of machine learning techniques for software fault prediction,” *Appl. Soft Comput.*, vol. 27, no. C, pp. 504–518, 2015.
- [161] G. McGraw, *Software Security: Building Security In*. Addison-Wesley Professional, 2006.

-
- [162] Y. Shin and L. Williams, “Is complexity really the enemy of software security?” in *QoP*, 2008, pp. 47–50.
- [163] —, “An empirical model to predict security vulnerabilities using code complexity metrics,” in *ESEM*, 2008, pp. 315–317.
- [164] I. Chowdhury and M. Zulkernine, “Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities,” *J. Syst. Archit.*, vol. 57, no. 3, pp. 294–313, 2011.
- [165] S. Moshtari, A. Sami, and M. Azimi, “Using complexity metrics to improve software security,” *Computer Fraud and Security*, vol. 2013, no. 5, pp. 8–17, 2013.
- [166] S. Moshtari and A. Sami, “Evaluating and comparing complexity, coupling and a new proposed set of coupling metrics in cross-project vulnerability prediction,” in *SAC*, 2016, pp. 1415–1421.
- [167] M. Gegick, L. Williams, J. Osborne, and M. Vouk, “Prioritizing software security fortification throughcode-level metrics,” in *QoP*, 2008, pp. 31–38.
- [168] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne, “Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities,” *IEEE Trans. Softw. Eng.*, vol. 37, no. 6, pp. 772–787, 2011.
- [169] Y. Shin and L. Williams, “Can traditional fault prediction models be used for vulnerability prediction?,” *Empirical Software Engineering*, vol. 18, no. 1, pp. 25–59, 2013.
- [170] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen, “Predicting vulnerable software components via text mining,” *IEEE Transactions on Software Engineering*, vol. 40, no. 10, pp. 993–1006, 2014.
- [171] J. Walden, J. Stuckman, and R. Scandariato, “Predicting vulnerable components: Software metrics vs text mining,” in *ISSRE*, 2014, pp. 23–33.
- [172] Y. Zhang, D. Lo, X. Xia, B. Xu, J. Sun, and S. Li, “Combining software metrics and text features for vulnerable file prediction,” in *ICECCS*, 2015, pp. 40–49.
- [173] A. Hovsepyan, R. Scandariato, and W. Joosen, “Is newer always better?: The case of vulnerability prediction models,” in *ESEM*, 2016, pp. 26:1–26:6.

- [174] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, “Predicting vulnerable software components,” in *CCS*, 2007, pp. 529–540.
- [175] V. H. Nguyen and L. M. S. Tran, “Predicting vulnerable software components with dependency graphs,” in *MetriSec*, 2010, pp. 3:1–3:8.
- [176] P. Morrison, K. Herzig, B. Murphy, and L. Williams, “Challenges with applying vulnerability prediction models,” in *HotSoS*, 2015, pp. 4:1–4:9.
- [177] Y. Shin and L. Williams, “An initial study on the use of execution complexity metrics as indicators of software vulnerabilities,” in *SESS*, 2011, pp. 1–7.
- [178] S. Son, K. S. McKinley, and V. Shmatikov, “Rolecast: Finding missing security checks when you do not know what checks are,” in *OOPSLA*, 2011, pp. 1069–1084.
- [179] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, “Chucky: Exposing missing checks in source code for vulnerability discovery,” in *CCS*, 2013, pp. 499–510.
- [180] J. Vanegue and S. K. Lahiri, “Towards practical reactive security audit using extended static checkers,” in *SP*, 2013, pp. 33–47.
- [181] F. Yamaguchi, M. Lottmann, and K. Rieck, “Generalized vulnerability extrapolation using abstract syntax trees,” in *ACSAC*, 2012, pp. 359–368.
- [182] V. B. Livshits and M. S. Lam, “Finding security vulnerabilities in java applications with static analysis,” in *USENIX Security*, 2005.
- [183] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck, “Modeling and discovering vulnerabilities with code property graphs,” in *SP*, 2014, pp. 590–604.
- [184] F. Camilo, A. Meneely, and M. Nagappan, “Do bugs foreshadow vulnerabilities?: A study of the chromium project,” in *MSR*, 2015, pp. 269–279.
- [185] T. J. McCabe, “A complexity measure,” *IEEE Trans. Softw. Eng.*, vol. 2, no. 4, pp. 308–320, 1976.
- [186] X. Xie, B. Chen, Y. Liu, W. Le, and X. Li, “Proteus: Computing disjunctive loop summary via path dependency analysis,” in *FSE*, 2016, pp. 61–72.

-
- [187] L. Szekeres, M. Payer, T. Wei, and D. Song, “Sok: Eternal war in memory,” in *SP*, 2013, pp. 48–62.
- [188] “Leopard,” <https://sites.google.com/site/leopardsite2017/>, 2017.
- [189] “Cvedetails,” <http://www.cvedetails.com/>, 2017.
- [190] “National vulnerability database,” <https://nvd.nist.gov/>, 2017.
- [191] Z. Xu, B. Chen, M. Chandramohan, Y. Liu, and F. Song, “Spain: Security patch analysis for binaries - towards understanding the pain and pills,” in *ICSE*, 2017.
- [192] (2017) American fuzzy lop. <http://lcamtuf.coredump.cx/afl/>.
- [193] J. Wang, B. Chen, L. Wei, and Y. Liu, “Skyfire: Data-driven seed generation for fuzzing,” in *SE&P*, 2017, pp. 579–594.
- [194] G. Coldwind, “How to find vulnerabilities?” <http://gynvael.coldwind.pl/?lang=en&id=659>, 2017.
- [195] “Flawfinder,” <https://www.dwheeler.com/flawfinder/>, 2017.
- [196] “Pscan,” <http://deployingradius.com/pscan/>, 2017.
- [197] “Rats,” <https://code.google.com/archive/p/rough-auditing-tool-for-security/>, 2017.
- [198] J. Viega, J. T. Bloch, Y. Kohno, and G. McGraw, “Its4: A static vulnerability scanner for c and c++ code,” in *ACSAC*, 2000, pp. 257–267.
- [199] J. R. Larus, T. Ball, M. Das, R. DeLine, M. Fahndrich, J. Pincus, S. K. Rajamani, and R. Venkatapathy, “Righting software,” *IEEE Softw.*, vol. 21, no. 3, pp. 92–100, 2004.
- [200] B. Hackett, M. Das, D. Wang, and Z. Yang, “Modular checking for buffer overflows in the large,” in *ICSE*, 2006, pp. 232–241.
- [201] L. Tan, X. Zhang, X. Ma, W. Xiong, and Y. Zhou, “Autoises: Automatically inferring security specifications and detecting violations,” in *USENIX Security*, 2008, pp. 379–394.

- [202] F. Yamaguchi, A. Maier, H. Gascon, and K. Rieck, “Automatic inference of search patterns for taint-style vulnerabilities,” in *SP*, 2015, pp. 797–812.
- [203] I. Medeiros, N. Neves, and M. Correia, “Dekant: A static analysis tool that learns to detect web application vulnerabilities,” in *ISSTA*, 2016, pp. 1–11.
- [204] D. Evans and D. Larochelle, “Improving security using extensible lightweight static analysis,” *IEEE Softw.*, vol. 19, no. 1, pp. 42–51, 2002.
- [205] M. Zitser, R. Lippmann, and T. Leek, “Testing static analysis tools using exploitable buffer overflows from open source code,” in *FSE*, 2004, pp. 97–106.
- [206] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner, “Detecting format string vulnerabilities with type qualifiers,” in *USENIX Security*, 2001.
- [207] Y. Xie and A. Aiken, “Static detection of security vulnerabilities in scripting languages,” in *USENIX Security*, 2006.
- [208] N. Jovanovic, C. Kruegel, and E. Kirda, “Pixy: A static analysis tool for detecting web application vulnerabilities (short paper),” in *SP*, 2006, pp. 258–263.
- [209] J. Dahse and T. Holz, “Simulation of built-in php features for precise static code analysis,” in *NDSS*, 2014.
- [210] S. Lekies, B. Stock, and M. Johns, “25 million flows later: Large-scale detection of dom-based xss,” in *CCS*, 2013, pp. 1193–1204.
- [211] P. Saxena, S. Hanna, P. Poosankam, and D. Song, “Flax: Systematic discovery of client-side validation vulnerabilities in rich web applications,” in *NDSS*, 2010.