Efficient, Scalable, and Accurate Program Fingerprinting in Binary Code

Saed Alrabaee

A Thesis

In

The Concordia Institute

for

Information Systems Engineering

Presented in Partial Fulfillment of the Requirements

For the Degree of

Doctor of Philosophy (Information and Systems Engineering) at

Concordia University

Montreal, Quebec, Canada

February 2018

CONCORDIA UNIVERSITY
SCHOOL OF GRADUATE STUDIES

This is to certify that the thesis prepared

By: Saed Alrabaee

Entitled: Efficient, Scalable, and Accurate Program Fingerprinting in Binary Code

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

Doctor Of Philosophy (Information and Systems Engineering)

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

Signed by the final examining committee:

Chair
Dr. Joey Paquet

External Examiner
Dr. Jean-Yves Marion

External to Program
Dr. Peter Grogono

Examiner
Dr. Amr Youssef

Examiner
Dr. Mohammad Mannan

Thesis Supervisor (s)
Dr. Mourad Debbabi

Dr. Lingyu Wang

Approved by

Dr. Rachida Dssouli Chair of Department or Graduate Program Director

April 06, 2018
Date of Defence

Dr. A. Asif, Dean, of Faculty of Engineering and Computer Science

ABSTRACT

Efficient, Scalable, and Accurate Program Fingerprinting in Binary Code

Saed Alrabaee, Ph. D.

Concordia University, 2018

Why was this binary written? Which compiler was used? Which free software packages did the developer use? Which sections of the code were borrowed? Who wrote the binary? These questions are of paramount importance to security analysts and reverse engineers, and binary fingerprinting approaches may provide valuable insights that can help answer them. This thesis advances the state of the art by addressing some of the most fundamental problems in program fingerprinting for binary code, notably, reusable binary code discovery, fingerprinting free open source software packages, and authorship attribution. First, to tackle the problem of discovering reusable binary code, we employ a technique for identifying reused functions by matching traces of a novel representation of binary code known as the semantic integrated graph. This graph enhances the control flow graph, the register flow graph, and the function call graph, key concepts from classical program analysis, and merges them with other structural information to create a joint data structure. Second, we approach the problem of fingerprinting free open source

software (FOSS) packages by proposing a novel resilient and efficient system that incorporates three components. The first extracts the syntactical features of functions by considering opcode frequencies and performing a hidden Markov model statistical test. The second applies a neighborhood hash graph kernel to random walks derived from control flow graphs, with the goal of extracting the semantics of the functions. The third applies the z-score to normalized instructions to extract the behavior of the instructions in a function. Then, the components are integrated using a Bayesian network model which synthesizes the results to determine the FOSS function, making it possible to detect user-related functions.

With these elements now in place, we present a framework capable of decoupling binary program functionality from the coding habits of authors. To capture coding habits, the framework leverages a set of features that are based on collections of functionality-independent choices made by authors during coding. Finally, it is well known that techniques such as refactoring and code transformations can significantly alter the structure of code, even for simple programs. Applying such techniques or changing the compiler and compilation settings can significantly affect the accuracy of available binary analysis tools, which severely limits their practicability, especially when applied to malware. To address these issues, we design a technique that extracts the semantics of binary code in terms of both data and control flow. The proposed technique allows more robust binary analysis because the extracted semantics of the binary code is generally immune from code transformation, refactoring, and varying the compilers or compilation settings.

Specifically, it employs data-flow analysis to extract the semantic flow of the registers as well as the semantic components of the control flow graph, which are then synthesized into a novel representation called the semantic flow graph (SFG). We evaluate the framework on large-scale datasets extracted from selected open source C++ projects on GitHub, Google Code Jam events, Planet Source Code contests, and students' programming projects and found that it outperforms existing methods in several respects. First, it is able to detect the reused functions. Second, it can identify FOSS packages in real-world projects and reused binary functions with high precision. Third, it decouples authorship from functionality so that it can be applied to real malware binaries to automatically generate evidence of similar coding habits. Fourth, compared to existing research contributions, it successfully attributes a larger number of authors with a significantly higher accuracy. Finally, the new framework is more robust than previous methods in the sense that there is no significant drop in accuracy when the code is subjected to refactoring techniques, code transformation methods, and different compilers.

ACKNOWLEDGEMENTS

I would like to express my heartfelt gratitude to my supervisors Prof. Mourad Debbabi and Prof. Lingyu Wang, who contributed to this thesis and improved it significantly with their guidance and advises . Their affluent and profound knowledge, precious insights, and constructive criticism allowed me to build a successful research. I greatly appreciate their dedication to helping students academically. I should not forget to mention that despite Dr. Debbabi had very busy schedule, he allocated time to meet all students. Moreover, his charisma inspires everyone. Thank you for everything! My gratefulness extends to the members of the examining committee: Drs. Jean-Yves Marion, Peter Grogono, Amr Youssef, and Mohammad Mannan, who honored me by accepting to evaluate this thesis. Their time and efforts are highly appreciated. Special thanks to my colleagues, namely, Paria Shirani, Noman Saleem, Stere Preda, Ashkan Rahimian, who provided their valuable expertise. I convey very special acknowledgements to Paria Shirani who provided me with insightful technical discussions. I would like also to thanks my close friend, Mahmoud Khasawneh, for stimulating discussions and good memories shared together. I feel very fortunate to have such nice friend. Further, I would like to express my gratitude to Abdullah Amareen, Momen Oqaily, Ahmad Bataineh, and Suhib Melhem, who supported me strongly. last but not least, I would like to express my profound gratitude to my beloved my parents, sisters, and brothers, who expressed their encouragement and love.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

Binary (i.e., executable) code fingerprinting is essential to many security applications; examples include malware detection, software infringement, vulnerability analysis, and digital forensics. Furthermore, it is useful for security researchers and reverse engineers since it offers very important insights of the binary code, such as revealing the functionality, authorship attribution, libraries used, and vulnerabilities. To this end, numerous studies focus on analyzing binary code with the goal of extracting fingerprints that can illuminate the semantics of a target application. However, extracting fingerprints is a challenging task since a substantial amount of important information will be lost during compilation, notably, variable and function naming, the original control and data flow structures, comments, semantic information, and the layout. This thesis advances the state of the art by addressing some of the most fundamental problems in program fingerprinting for binary code, notably, reusable binary code discovery, fingerprinting free open

source software packages, and authorship attribution.

1.1 Motivations

Reused Function Identification. The objective of reverse engineering often involves understanding both the control and data-flow structures of the functions in the given binary code. However, this is usually a challenging task, as binary code inherently lacks structure due to the use of jumps and symbolic addresses, highly optimized control flow, varying registers and memory locations based on the processor and compiler, and the possibility of interruptions [36]. To assist reverse engineers in such a difficult task, automated tools for efficiently recognizing reused functions for binary code are highly desirable. This is especially true in the context of malware analysis, since modern malware are known to contain a significant amount of reused code derived from previous existing code [38,117].

Free Open Source Packages Identification. When analyzing malware binaries, reverse engineers often pay special attention to reused free open source packages for several reasons. First, recent reports from anti-malware companies indicate that finding the similarity between malware codes attributable to reused third-party libraries can aid in developing profiles for malware families [85]. For instance, `Flame` [39] and other malware in its family [39] all contain code packages that are publicly available, including SQLite and LUA [85]. Second, a significant proportion of most modern malware consists of third-party libraries; as such, identifying reused libraries is a critical preliminary step in the process of extracting information about the functionality of a malware binary. Third,

in more challenging cases where obfuscation techniques may have been applied and the reused third-party libraries may differ from their original source files, it is still desirable to determine which part of the malware binary is borrowed from which third-party libraries. Fourth, in addition to identifying third-party libraries, clustering third-party libraries based on their common origin may help reverse engineers to identify new malware from a known family or to decompose a malware binary based on the origin of its functions. Fifth, third-party libraries should be filtered out when the authorship attribution tools are applied.

Binary Authorship Identification. Existing approaches to binary authorship attribution typically employ machine learning methods to extract unique features for each author and subsequently match a given binary against such features to identify the author [27, 44, 112]. We have studied and analyzed these approaches in previous research [28]. We have found that these approaches also share a critical limitation: they cannot distinguish between features related to author style (e.g., coding habits) and features related to functionality. Consequently, the extracted features, though unique for each author, may be completely unrelated to programming style. In addition, other limitations, such as a significantly lower accuracy in the case of multiple authors, being easily defeated by refactoring techniques or simple obfuscation methods, and not being validated against real malware, are also shared by these existing efforts. More recently, the feasibility of authorship attribution on malware binaries was discussed at the BlackHat conference [91]. A set of features are employed to group malware binaries according to authorship [91].

However, the process is not automated and requires considerable human intervention. We present a component that is designed to recognize authors' *coding habits* by decoupling them from program functionality in binary code. Instead of using generic features (e.g., n-gram or small subgraphs of a CFG [112]), which may or may not be related to authorship, *BinAuthor* captures coding habits based on a collection of functionality-independent choices frequently made by authors during coding (e.g., preferring to use either `if` or `switch`, and relying more on either object-oriented modularization or procedural programming).

1.2 Research objectives and contributions

The main goals of our research are summarized in the following points:

- **Authorship Analysis**. Malware analysts are typically interested to discover clues that lead to the parties that are responsible. Such clues should be able to discriminate code written by different developers, which also might be used to discover stylistic similarities between binary programs.

- **Function Fingerprinting**. Fingerprints are useful in automating reverse engineering tasks including clone detection, library identification, authorship attribution, cyber forensics, etc. In this thesis, a set of tools are designed for fingerprinting binary functions. The main objective is to provide an accurate and scalable solution to fingerprint (i) reused binary functions; and (ii) reused free open source packages.

Specifically, this thesis makes the following contributions:

1.2.1 *SIGMA*: Reused Function Identification [29]

This component is based on a technique for identifying reused functions in binary code by matching traces of a novel representation of binary code, namely, the Semantic Integrated Graph (*SIG*). The *SIG* enhances and merges several existing concepts from classic program analysis, including control flow graph, register flow graph, and function call graph into a joint data structure. Such a comprehensive representation allows us to capture different semantic descriptors of common functionalities in a unified manner as graph traces, which can be extracted from binaries and matched to identify reused functions, actions, or open source software packages. In summary, our contributions to the problem of identifying reused functions in binary code are as follows:

- We introduce the novel *SIG* representation of binary code to unify various semantic information, such as control flow, register manipulation, and function call into a joint data structure to facilitate more efficient graph matching.

- We define different types of traces such as normal traces, AND-traces, and OR-traces over *SIG* graphs, which are used for inexact matching. We carry out both exact and inexact matching between different binaries, where an exact matching applies to two *SIG* graphs with the same graph properties (e.g. number of nodes), whereas an inexact matching employs graph edit distance to measure the degree of similarity between two *SIG* graphs of different sizes.

1.2.2 *FOSSIL*: Free Open Source Packages Identification [30]

This component is based on a novel resilient and efficient system that incorporates three layers. The first layer extracts the syntactical features of functions by considering opcode frequencies and applying a hidden Markov model statistical test. The second layer applies a neighborhood hash graph kernel to random walks derived from control flow graphs, with the goal of extracting the semantics of the functions. The third layer applies z-score to the normalized instructions to extract the behavior of instructions in a function. The layers are integrated using a Bayesian network model which synthesizes the results to determine the FOSS function. The novel approach of combining these layers using the Bayesian network has produced stronger resilience to code obfuscation. In short, it makes the following contributions:

- *FOSSIL* is the first system developed to identify reused FOSS packages in malware binaries that supports multiple feature (syntactic, semantic, and structural features). Its novelty also lies in its ability to integrate the ranked opcodes, subgraph search, and function behavior. This helps reverse engineers to recognize the types of applications that a malware binary incorporates in order to characterize the malware.

- We propose an adaptive hidden Markov and Bayesian model capable of approximating the similarity between functions. This adaptive model boosts the matching search quality and yields stable results across different datasets and metrics.

1.2.3 *BinAuthor*: Binary Authorship Attribution

This component is based on a collection of functionality-independent choices frequently made by authors during coding (e.g., preferring to use either `if` or `switch`, and relying more on either object-oriented modularization or procedural programming). Our main contributions are as follows:

- To the best of our knowledge, this component is the first system capable of decoupling author coding habits from program functionality in binary code. The novel approach of using functionality-independent features allows our system to overcome a key limitation of most existing works: assuming the existence of special training data (binaries with identical functionality but written by different authors [27, 44, 112]). By avoiding such an unrealistic assumption, our system paves the way towards practical applications of automated binary authorship attribution.

- The proposed system yields high accuracy that survives refactoring and source/binary obfuscation techniques. This shows the potential of our system as a practical tool to assist reverse engineers in a number of security-related tasks, such as identifying the author of a malware sample, clustering malware samples based on common authors, and determining the number of authors (e.g., a large number of authors may indicate an organizational effort).

1.2.4 *BinGold*: Extracting the Semantics of Binary Code [31]

This component is based on extracting various types of semantics and integrating them into a novel representation called a Semantic Flow Graph (SFG). This component makes following contributions:

- We introduce the novel SFG representation of binary code to unify various semantic information, such as control flow, register manipulation, data flow analysis, and function call into a joint data structure to facilitate more efficient graph matching.

- We define different types of traces over SFG graphs to serve as matching features and then carry out both exact and inexact matching between different binaries, where an exact matching applies to two SFG graphs with the same number of nodes, and an inexact matching employs graph edit distance to measure the degree of similarity between two SFG graphs of different size.

- We test our method on a large test suite across different operating systems, compilers, and compiling optimizations. Our results show that our method achieves higher accuracy than previously available fingerprint representations.

1.3 Thesis Organization

The remainder of this thesis is structured as follows: In Chapter 2, we present the background literature about binary analysis and review the state-of-the-art techniques that are

close to the proposed framework. In Chapter 3, we introduce our system for identifying reuse functions in binary code. Chapter 4 provides our designed system for fingerprinting free open source code. In Chapter 5, we introduce the binary authorship attribution. In Chapter 6, we detail the contribution of our framework in extracting the semantics of binary code. Chapter 7 provides concluding remarks together with a discussion of future works.

Chapter 2

Background and Related Work

This thesis investigates different aspects of binary code, the most important question of which is how to extract particular aspects from a binary code. This question is central to many works on binary code analysis. Therefore, it is necessary to understand the efforts that are most closely related to the applications of our framework. In this chapter, we review studies that extract semantics characteristics from binary programs. Specifically, this chapter first presents an overview of the importance of binary analysis. Then, it highlights the challenges might be received. After that, it reviews three areas of related work. First, it introduces the existing efforts in reused code identification. Subsequently, it highlights the existing efforts that are related to fingerprinting free open source software packages. Finally, we highlight the existing techniques for authorship attribution.

2.1 Importance of Binary Analysis

Binary code analysis is an important process in many security applications such as malware analysis [37, 57, 95, 128], software reliability [89], reverse engineering [42, 61], debugging [33], digital forensics [27], and security analysis [62, 73, 103].

2.1.1 Reverse Engineering

Binary code analysis is considered as the crucial process for the reverse engineers in several tasks: (i) Authorship attribution that refers to the process of identifying the author of an anonymous binary file based on stylistic characteristics. Its aim is to automate the laborious and error-prone reverse engineering task of discovering information related to the author(s) of binary code [28]. (ii) Reused code discovery is the process of determining the reused free open-source software packages [102, 115]. (iii) Compiler provenance identification encompasses numerous pieces of information, such as the compiler family, compiler version, optimization level, and compiler-related functions [108, 114]. (iv) Program binaries normally contain a significant amount of third-party library functions taken from standard libraries and the process of determining such functions is called library functions identification.

2.1.2 Malware Analysis

Binary analysis is important for malware analysis because when malware attacks a computer system or network, it leaves an executable behind but rarely the source code. A

comprehensive understanding of binary analysis is particularly vital in the ongoing warfare between malware writers and anti-malware vendors. Each camp performs different operations on the binary code according to their goals as described below.

- Binary modification enables a malware writer to alter existing code or to inject new code into programs to execute malicious behavior without requiring the source code or debugging information [93, 118].

- Binary comparison automatically classifies new malware samples based on the assumption that unknown malware is often produced from known malware [35].

- Binary obfuscation is used by malware writers to obfuscate malicious code to avoid existing detection techniques, such as misuse detection algorithms. This operation relies on the fact that the detection techniques are based on sensitivity to slight modifications in the program syntax [130].

- Virus signatures are used to identify specific code patterns, called signatures, within a program. When a signature is found, the program most likely contains a virus. For instance, scanning for the following hexadecimal sequence can identify the Chernobyl virus [124]:

  ```
  E800 0000 005B 8D4B 4251 5050

  0F01 4C24 FE5B 83C3 1CFA 8B2B
  ```

  In response, malware writers may attempt to complicate the binary code to thwart

anti-virus techniques. Although the signature identification method is highly effective in identifying known malware, it cannot identify new or unknown malware [70].

The evasion techniques are summarized below.

- Compression techniques compress the executable binary file and include a decompression algorithm in the code, which enables the compressed program to run [118].
- Polymorphism is a technique that produces different packed binary file versions from the same source input [121].
- Code obfuscation alters malicious code to help it avoid detection. Obfuscation can thwart most existing detection techniques [130].
- Packing is a method used by malware writers to hide their software from signature-based investigation techniques. While packing techniques vary substantially, their objectives are identical: to modify the appearance of the program code while ensuring that the semantics remain the same. A common approach is to use binaries created directly in assembly language rather than compiled from a high-level language to avoid any evidence of the author's intentions [80].

2.1.3 Digital Forensics

FireEye [97] discovered that malware binaries share very important information that reveal the digital infrastructures used, code traits, and other semantic information, such as

timestamp, the use of certificates, executable resources and development tools. FireEye investigators eventually noticed that malware binaries of the same, previously discovered, infrastructures are written by the same group of authors. In such cases, training on such binaries and some random authors' code may offer a vital help to forensics investigators. In addition, testing a recent piece of malware binary code using some confidence metrics would verify if a specific author is the actual author.

2.1.4 Software Infringement

It is very important to analyze the binary code in order to discover that a piece of code is not written by the claimed author. Generally, the adversary attempts to modify code written by another author to match his/her own style [43, 44]. In forensics applications, two of the parties may collaborate to modify the style of code written by one to match the other's style [43,44]. This emphasizes the importance of binary analysis to discover clues that help in tackle such scenarios. Also, building an online repository of candidate authors based on previously collected malware samples, would greatly help in infringement analysis [43].

2.2 Binary Analysis Challenges

Binary code analysis poses a considerably greater challenge than source code analysis [93]. These challenges are summarized as follows:

C1 **Compiler's effects**. Compilers may have inserted substantial changes in the binary code such as compiler tags [114]. These changes tend to hinder the abilities of binary analysis approaches, causing inaccurate reported results produces by approaches. Such coding changes influence the analytical aptitude for understanding the processes and objectives of a program and hinder the approach's ability to appropriately analyze the binary program.

C2 **Lack of semantics**. In essence, binary code lacks the larger amount of semantic information available in source code. Such information may be related to the code structure, the buffer characteristics, and the function prototypes that are to be executed at the binary stage, complicating the binary analysis.

C3 **Function boundary identification**. Most of existing efforts disassemble the binary file into assembly file for gaining more information. This is considered as the first step in binary analysis. Assembly file encompasses a set of functions. Such functions should be recognized by defining the starting and ending address of each function [32]. However, many existing tools cannot recognize a function's starting point [116]; therefore, they are unable to access the actual code. Therefore, the many binary analysis techniques which rely on function boundary information must first attempt to recover it through function identification.

C4 **Binary format**. For an x86 computer processor, the familiar formats currently include the *Executable and Linking Format* (ELF) and the *Portable Executable* (PE)

format employed in Linux and Windows, respectively. These two formats divide a file into segments (i.e., data and control) that can be allocated to store and/or code data. The segments can be identified as writable, readable, and/or executable at runtime. However, no guidelines exist regarding the allotment of data and code. For instance, code sections frequently incorporate data such as string constants or jump tables. Identifying all compiled binary segments as writable, readable, or executable may not be possible.

C5 **Code Sections discovery**. Binary code encompasses two main sections: code and data sections. The segregation process between these sections is very important since binary analysis approaches may misinterpret critical data bytes as instructions or miss real instructions [93]. Also, modifying the critical data will cause crashing the program. Moreover, the binary fingerprinting approaches require the data sections ro be recognized carefully in order to extract a set of efficient features for their purposes.

C6 **Debug information availability**. When debugging information is stripped from the binary code, much of the valuable information will be lost in stripping process. These information include the strings, variable information, and the standard library functions linked into the binary code [74]. This hinders the binary fingerprinting approaches.

2.3 Binary Code Transformation

This section describes a set of binary code transformation methods that most existing works, especially for fingerprinting applications (e.g., authorship), could be affected by them. These methods include certain binary disturbances such as compiler optimizations, differences in build environments, refactoring process, etc. In what follows, we describe examples of such disturbances.

2.3.1 Function Inlining

In practice, the compiler may inline a small function into its caller code as an optimization. This may introduce additional complexity to the code. Furthermore, function-inlining significantly changes the CFG of a program, which may become problematic for existing binary analysis approaches [106]. Finding inlined code is a challenging task [107]. The accuracy will undoubtedly drop if the features are derived from a function that includes inlined functions or if the target programs do not show such inlining. Still, using the multiple initial basic block matches will not likely find the multiple counterparts in the non-inlined target program [106].

2.3.2 Instruction Reordering

Compilers may reorder independent computations to enhance data locality. Reordered instructions in a basic block change the syntactic representation [106]. However, the semantics of a basic block remain the same.

2.3.3 Refactoring Process

Refactoring process might alter the structure of code without changing the way it behaves [67]. Refactoring is considered a best practice when creating and maintaining software; indeed, research suggests that programmers practice it regularly [99, 125]. Examples of refactoring include renaming a variable, moving a method from a superclass to its subclasses, and taking a few statements and extracting them into a new method. These examples are referred to as `RENAME, PUSH DOWN METHOD, and EXTRACT METHOD` [67].

2.4 Reused Function Identification

There are several frameworks proposed for extracting the semantics of binary code for particular tasks, such as BinSlayer [41], BinJuice [86], BitShred [75], and iBinHunt [96]. Such frameworks may use for identifying reused functions. BinSlayer uses a polynomial algorithm to find the similarity between executables, obtained by fusing the well-known BinDiff algorithm [66] with the Hungarian algorithm [98] for bi-partite graph matching. BinJuice extracts the abstraction of the semantics of binary blocks which is termed "juice". Whereas the denotational semantics summarizes the computation performed by a block, its juice presents a template of the relationships established by the block [86]. BitShred is a framework for automatic code reuse detection in binary code [75]. BitShred can be used for identifying the amount of shared code based on the ability to calculate

the similarities among binary code. iBinhunt is a technique to find the semantic differences between two binary programs when the source code is not available. It uses the process of analyzing control flow, particularly intra-procedural control flow [96]. More recently, `ESH` a new tool has been proposed [52]. They introduce a statistical approach for measuring the similarity between two fragments (e.g., procedures). Genius [64] employs different set of features, generates codebooks from CFGs, translates the codebooks into numeric vectors, and finally using locality sensitive hashing (LSH) to overcome the scalability.

2.5 Fingerprinting Free Open Source Function

In this section, we review relevant research works on binary function fingerprinting.

2.5.1 Search-Based Function Fingerprinting

Creating a search engine for executables is an extremely important issue, as it helps reverse engineers to detect the functionality of the code. To the best of our knowledge, there are three search engines for binaries: "Rendezvous" [81], "Tracelet" [53], and "SARVAM" [101]. The SARVAM search engine is designed for malware binaries. Given a malware query, a fingerprint is first computed based on transformed image features [101]. Tracelet introduces an engine for searching binary functions in the code base. The authors decompose CFGs into fixed length subtraces excluding jump instructions, which are called `tracelets`. Rendezvous [81] identifies binary code using a statistical model

comprising instruction mnemonics, control flow sub-graphs, and data constant features. BinClone [63] is a binary clone system that shows the feasibility of detecting exact clones in assembly code based on n-grams. However, it suffers from signature collisions and is not scalable. Most recently, BinGo [47] a cross-architecture binary code search is proposed. This system consists of two components: The first is designed to filter out OS functions and the second is to model binary function by extracting variant traces. A very recent approach namely BinSequence [72] has been proposed. They compare two functions by extracting the longest common sequence path and applying neighborhood exploration.

2.5.2 Dynamic Function Fingerprinting Methods

Dynamic fingerprinting methods are a set of ways to analyze a program to determine the specific inputs which cause each part of a program to execute. The work introduced by Homan et al. [76], the authors compared execution traces using longest common sequences. A new method to automatically find vulnerabilities and generate exploits has been proposed [34]. These authors propose preconditioned symbolic execution, a new technique for targeting symbolic execution. A new method is proposed for mutating well-formed program inputs or simply fuzzing, which is a highly effective and widely used strategy to find bugs in software [110]. Moreover, a binary search engine called Blanket Execution (BLEX) [58], executes functions and collects the side effects of functions; two functions with similar side effects are claimed to be similar.

2.6 Authorship Attribution

In this section, we review the related work to our main component that is designed for binary authorship attribution.

2.6.1 Source Code Authorship

Most previous studies of authorship analysis for general-purpose software have focused on source code [43, 83, 119]. These techniques are typically based on programming styles (e.g., naming variables and functions, comments, and code organization) and the development environment (e.g., OS, programming language, compiler, and text editor). The features selected by these techniques are highly dependent on the availability of the source code, which is seldom available when dealing with malware binaries. When dealing with executable files, it is infeasible to use most of these features as they are lost after the compilation and linking process. Spafford and Weeber [119] suggest that the use of lexical features (e.g., variable names) and syntactic features (e.g., usage of keywords) could aid in source code authorship attribution. Krsul and Spafford [83] attempt to find characteristics that represent coding style, suggesting that identifying programming style should be possible within a closed environment. More recently, Caliskan-Islam et al. [43] investigate methods to deanonymize source code authors of C++ using coding style.

2.6.2 Binary Code Authorship

In contrast to source code, binary code has drawn significantly less attention with respect to authorship attribution. This is mainly due to the fact that many salient features that may identify an author's style are lost during the compilation process. In [27, 44, 112], the authors show that certain stylistic features can indeed survive the compilation process and remain intact in binary code, thus showing that authorship attribution for binary code should be feasible. The methodology developed by Rosenblum et al. [112] is the first attempt to automatically identify authors of software binaries. The main concept employed by this method is to extract syntax-based features using predefined templates such as idioms (sequences of three consecutive instructions), *n*-grams, and graphlets. A subsequent approach to automatically identify the authorship of software binaries is proposed by Alrabaee et al. [27]. The main concept employed by this method is to extract a sequence of instructions with specific semantics and to construct a graph based on register manipulation. A more recent approach to automatically identify the authorship of software binaries is proposed by Caliskan-Islam et al. [44]. The authors extract syntactical features present in source code from decompiled executable binaries. Although these approaches represent solid progress on authorship attribution, not one of these approaches is applied to real malware, mostly due to their dependency on training data with the same functionality (which is infeasible in the case of malware). In [94], the authors introduce new fine-grained techniques to address the problem of identifying the multiple authors of binary code by determining the author of each basic block. They extract syntactic and

semantic features at a basic level, such as constant values in instructions, backward slices of variables, and width and depth of a function control flow graph (CFG). Furthermore, these approaches suffer from certain limitations, including low accuracy in the case of multiple authors and being potentially thwarted by simple obfuscation.

Moreover, most existing work on malware authorship attribution relies on manual analysis. In 2013, a technical report published by FireEye [97] discovered that malware binaries share the same digital infrastructure and code; for instance, the use of certificates, executable resources, and development tools. More recently, the team at Citizen Lab [11] attributed malware authors according to the manual analysis exploit type found in binaries and the manner by which actions are performed, such as connecting to a command and control server. The authors in [91] presented a novel approach to creating credible links between binaries originating from the same group of authors [91]. Their goal was to add transparency in attribution and to supply analysts with a tool that emphasizes or denies vendor statements. The technique is based on features derived from different domains, such as implementation details, applied evasion techniques, classical malware traits, or infrastructure attributes, which are leveraged to compare the handwriting among binaries.

We compare the existing authorship in terms of extracted features, implantation setup, and code availability, as shown in Table 2.1.

The extracted features include syntactic, structural, and semantic features. Most of the approaches are compatible with Linux binaries (e.g., ELF), and the binaries that they handle were originally compiled from C/C++ source code. Each approach requires a

Table 2.1: Comparison between different systems that identify the author of program binaries

Approach	Features			Implementation			Availability
	Syntax	Semantics	Structure	Language	Platforms	Dataset	
[27]	✓	✓	✗	C++	Windows	✓	Private
[44]	✓	✗	✓	C	Windows/Linux	✓	GitHub
[112]	✓	✓	✓	C++	Linux	✓	Public
[94]	✗	✓	✓	C++	Linux		Private

training dataset, with the exception of Meng's approach [94]. When the tools are available for researchers, we use private repositories, public repositories (indicates that the code is available but not included in general repositories), and general repositories (e.g., GitHub). Additionally, we categorize the existing approaches in Figure 2.1.

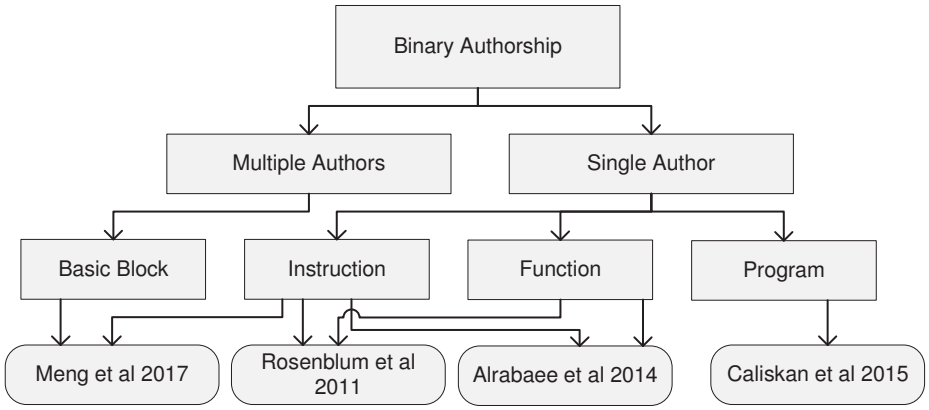


Figure 2.1: Taxonomy of authorship approaches

The features are extracted from different levels; for instance, Meng extracts features at the basic block level [94]. The machine learning algorithms employed also vary; for example, Rosenblum et al. used a support vector machine (SVM) algorithm [112],

whereas Caliskan et al. used a random forest classification algorithm [44].

2.7 Summary

In this chapter, we have reviewed different fields for the purpose of analyzing binary code. We initially investigated different aspects to highlight the importance of binary code analysis. Then, we explored certain challenges posed by the binary code analysis process. Such challenges should be considered by security researchers at any time they intend to design a tool for binary code analysis. Also, to tackle the drawbacks of aforementioned existing works, we design a framework that performs four tasks: identifying the compiler-related functions (this component is named *BinComp*), determining the reused functions (*SIGMA*), fingerprinting free open source software packages third-party libraries (*FOSSIL*), and finally attributing the author of program binaries (*BinAuthor*). Following chapters describe each component in details.

Chapter 3

Towards Identifying Reused Functions in Binary Code

3.1 Overview

The capability of efficiently recognizing reused functions for binary code is critical to many digital forensics tasks, especially considering the fact that many modern malware typically contain a significant amount of functions borrowed from open source software packages. Such a capability will not only improve the efficiency of reverse engineering, but also reduce the odds of common libraries leading to false correlations between unrelated code bases.

In this chapter, we propose *SIGMA*, a technique for identifying reused functions in

binary code by matching traces of a novel representation of binary code, namely, the semantic integrated graph (SIG). The SIGs enhance and merge several existing concepts from classic program analysis, including control flow graph, register flow graph, function call graph, and other structural information, into a joint data structure. Such a comprehensive representation allows us to capture different semantic descriptors of common functionalities in a unified manner as graph traces, which can be extracted from binaries and matched to identify reused functions, actions, or open source software packages.

3.2 Existing Representations of Binary Code

Numerous representations of binary code have been developed for different purposes of program analysis, such as data flow analysis, control flow analysis, call graph analysis, structural flow analysis, register manipulation analysis, and program dependency analysis. While these representations have been designed primarily for analyzing binary code, they can certainly be employed to characterize the code. In particular, we focus on three representations that capture structural information, namely, control flow graph, register flow graph, and function call graph. These representations form the basis of our approach to identifying reused functions in binary code. For the sake of clarity, we introduce a running example to illustrate these representations using the following sample code (bubble sort).

Listing 3.1: Bubble Sort C++ Program

```
 void bubble_sort(int arr[], int size) {
 bool not_sorted = true;
 int j=0,tmp;
while (not_sorted)
  {
   not_sorted = false;
   j++;
   for (int i = 0; i < size - j; i++){
       if (arr[i] > arr[i + 1]) {
         tmp = arr[i];
         arr[i] = arr[i + 1];
         arr[i + 1] = tmp;
         not_sorted = true;
       }//end of if
      print_array(arr,5);
    }//end of for loop
  }//end of while loop
}//end of bubble_sort
```

3.2.1 Control Flow Graph

Control flow graphs (CFGs) have been used for a variety of applications, e.g., to detect variants of known malicious applications [46]. A CFG describes the order in which basic block statements are executed as well as the conditions that need to be met for a particular path of execution. To this end, basic blocks are represented by nodes connected by

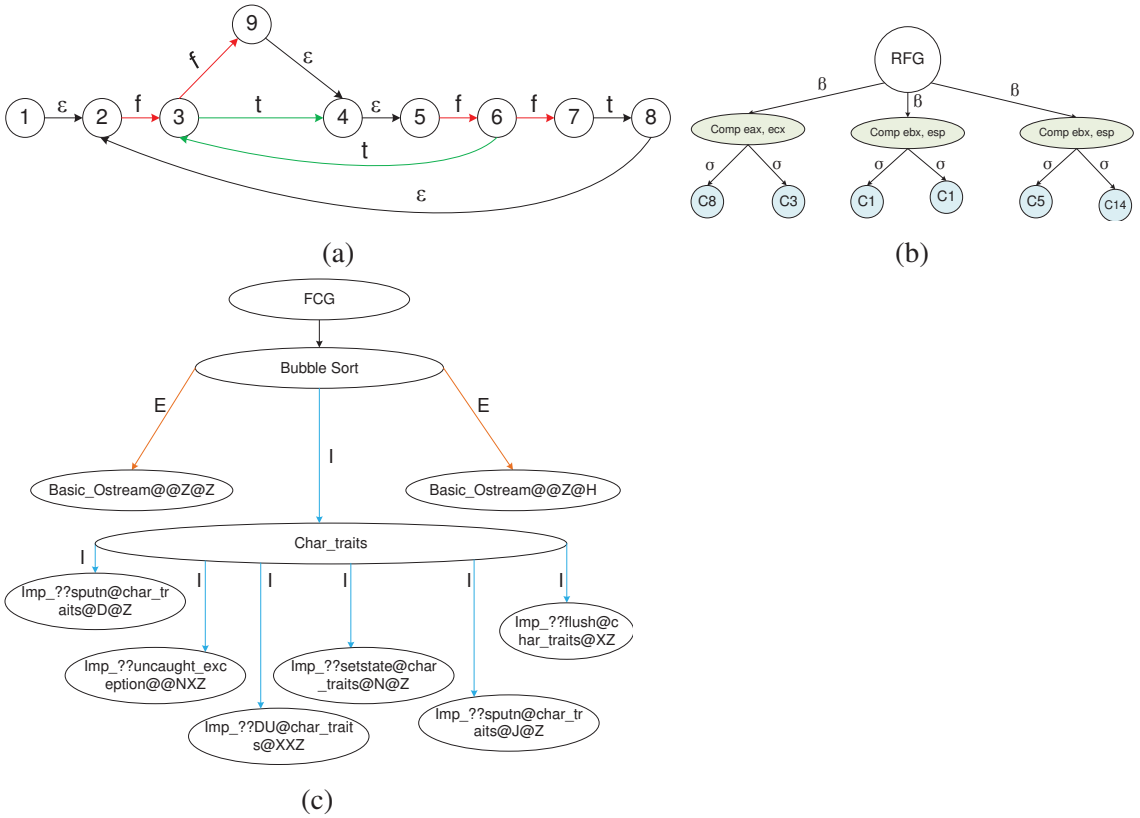


Figure 3.1: Classical representations for bubble sort function: (a) Control Flow Graph for bubble sort (b) Register Flow Graph for bubble sort function (c) Function Call Graph for bubble sort function

directed edges to indicate the transfer of control. It is necessary to assign a label `true (t)`, `false (f)`, or ϵ to each edge. In particular, a normal node has one outgoing edge labeled ϵ, whereas a predicate node has two outgoing edges corresponding to a true or false evaluation of the predicate. As an example, the CFG for bubble sort is shown in Figure 3.1(a). In our context, CFG is a standard code representation in reverse engineering to aid in understanding. However, while CFGs expose the control flow of a given code, they fail to provide other useful information, such as the way registers are manipulated by the code and the interaction between different functions.

3.2.2 Register Flow Graph

A register flow graph (RFG) is used to capture how registers are manipulated by binary code, which is originally designed for authorship identification of binary code [27]. RFGs describe the flow and dependencies between registers as an important semantic aspect of the behavior of a program. We briefly review the concept through an example shown in Figure 3.1(b). In the RFG, two costs are assigned to edges; β represents the basic block to which the compare instruction belongs (basic block id), and σ is the cost that is assigned based on the flow of the register's values (instruction counts). Regardless of the number or complexity degree, of functions, the following registers are often accessed: `ebp`, `esp`, `esi`, `edx`, `eax`, and `ecx`. Therefore, the steps involved in constructing a RFG for these registers are as follows:

- Counting the number of compare instructions,
- Checking the registers for each compare instruction,
- Checking the flow of each register from the beginning until the compare is reached,
- Classifying the register changes according to the 16 proposed classes in [27].

In RFGs, assembly instructions are classified into four families: stack, arithmetic, logical operation, and generic operation, as detailed in the following.

- Arithmetic: this class contains the following; `add`, `sub`, `mul`, `div`, `imul`, `idiv`, etc.
- Logical: this class contains the following; `or`, `and`, `xor`, `test`, `shl`.
- Generic: this class contains the following; `mov`, `lea`, `call`, `jmp`, `jle`, etc.
- Stack: this class contains `push` and `pop`.

3.2.3 Function Call Graph

A function call graph (FCG) is the representation of a function in binary code as a directed graph with labeled vertices, where the vertices correspond to functions and the edges to function calls. Two labels, `I` and `E` are assigned to the nodes; `I` represents internal library functions and `E` represents external library functions. An example of FCG for the bubble function is shown in Figure 3.1(c). In the literature, external call graphs have been used for malware detection [60]. In such a case, model graphs and data graphs are compared in order to distinguish call graphs representing benign programs from those based on malware samples [60, 111].

3.3 *SIGMA* Approach

In this section, we first provide an overview of the proposed *SIGMA* approach in Section 3.3.1. We then describe the three building blocks of an SIG in Section 3.3.2. We introduce the SIG concept in Section 3.3.3. Finally, we describe methods for SIG graph matching in Section 3.3.4.

3.3.1 Overview

The overall architecture of our *SIGMA* approach is depicted in Figure 3.2. There are two main phases: (i) a training phase, and (ii) a testing phase, detailed as follows.

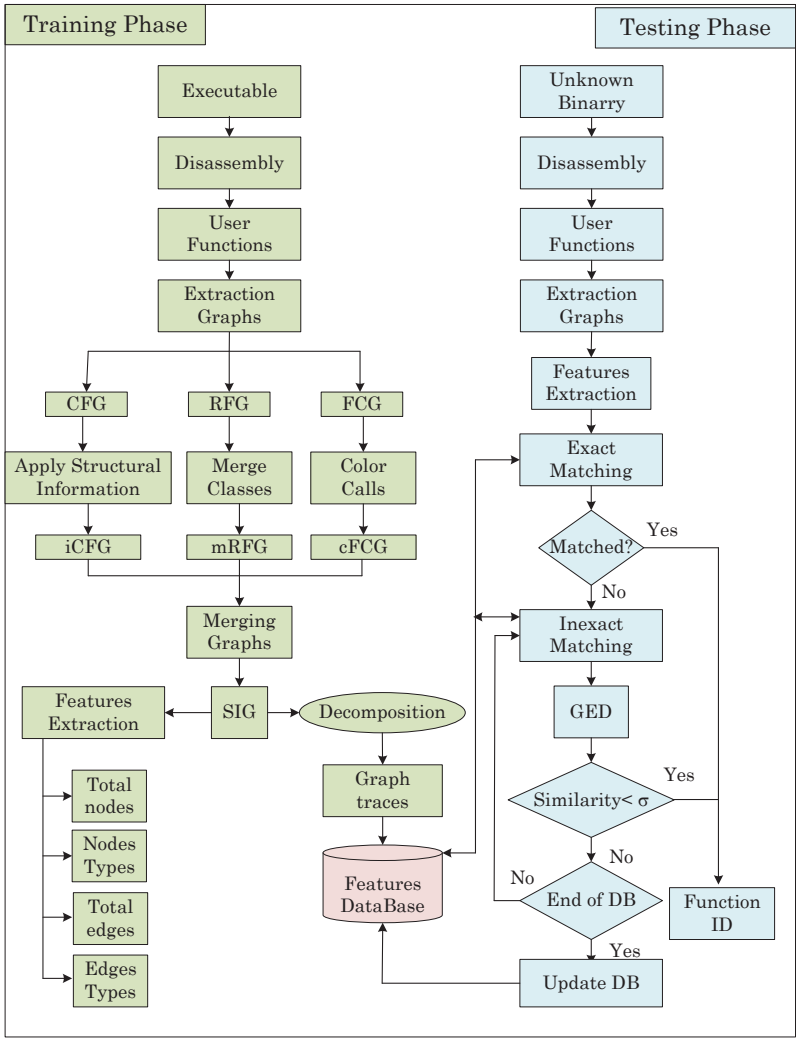


Figure 3.2: *SIGMA* architecture

The training phase consists of four steps; (i) disassembling the executable and manually filtering out functions related to the compiler; (ii) constructing the following graphs from user functions: CFG, RFG, and FCG; (iii) applying structural information to CFG to obtain the structural control flow graph $iCFG$; applying new merged classes to RFG to obtain a merged register flow graph $mRFG$; applying colored classes to FCG to obtain a colored function call graph $cFCG$ (those concepts will be explained in Section 3.3.2).

(iv) Merging the previous graphs into a single SIG. We then decompose the SIG into a set of traces aiming to identify fragments in the functions. Moreover, we consider various properties of the SIG, such as the total number of nodes, node types (data, control, dependence, or structural), edge types, total number of edges, the depth of the graph, etc. We save these details into a database with the function ID. On the other hand, given a set of unknown assembly instructions, the testing phase construct the SIG and extract the properties of the constructed graph and compare it with the existing SIGs graphs in the database. Hence, we have two methods for matching graphs: (i) *exact matching*: two graphs are said to match exactly if they have the same properties. Furthermore, a specific fragment g_i is said to belong to a specific function f_i if and only if f_i has a fragment with the same properties as g_i. (ii) *Inexact matching*: it is based on edit distance calculation and the result is compared to predefined threshold value α. Two functions are said to be the same if their similarity score is less than α. More formally, we have following definitions.

Definition 1. Let $\texttt{f}_1$, $\texttt{f}_2$ be two functions, we say $\texttt{f}_1$ is the copy (or origin) of $\texttt{f}_2$, if $\texttt{SIG(f}_1\texttt{)}$ matches $\texttt{SIG(f}_2\texttt{)}$.

Definition 2. Let $\texttt{f}_1$, $\texttt{f}_2$ be two functions, and $\texttt{SIG(f}_1\texttt{)} \longrightarrow \texttt{a}$ and let $\texttt{SIG(f}_2\texttt{)} \longrightarrow \texttt{b}$ denote extracting SIG traces $\texttt{a}$ and $\texttt{b}$ from $\texttt{f}_1$ and $\texttt{f}_2$. Let $\texttt{sim(a,b)}$ be a similarity function and δ a predefined threshold value ($\delta < 1$). We say $\texttt{f}_1$ and $\texttt{f}_2$ are similar if $1 - sim(a, b) < \delta$.

3.3.2 Building Blocks

In this section, we extend the existing representations introduced in Section 3.2 to form the building blocks of SIG.

A. Structural Information Control Flow Graph

As mentioned in Section 3.2, traditional CFGs consist of basic blocks each of which is a sequence of instructions terminating with a branch instruction. We can thus only obtain the structure of a function from a CFG. The lack of more detailed information in CFGs means two entirely different functions may yield the same CFG, which will cause confusion for identifying similar functions. Therefore, we extend standard CFGs with a colored scheme based on structural information about the probable role or functionality of each node. For example, if the majority of instructions in one node is arithmetic or logical, it may provide hints about the functionality of the node (e.g., cryptographic function usually involves a large number of `for` loops). By enriching standard CFGs with such information as different colors of nodes, which we call $iCFG$, we have a better chance to distinguish two functions even if they have the same CFG structure. Table 3.1 shows some example categories of structural information we consider in coloring the nodes.

The assignment of classes depends on two percentages: (i) the two highest percentages, and (ii) the lowest percentage, among the proposed categories. By considering the highest percentages, we aim to measure the majority category in the function. We choose two highest percentages because we have noticed that some classes, such as

Table 3.1: Structural information categories

Category	Description
Data Transfer (DT)	Data transfer instructions such as `mov, movzx, movsx`
Test(T)	Test instructions such as `cmp, test`
ArLo	Arithmetic and logical instructions such as `add, sub, mul, div, imul, idiv, and, or, xor, sar, shr`
CaLe	System call, API call, and Load effective instructions such as `lea`
Stack	Stack instructions such as `push, pop`

`Data Transfer`, are always dominant in many cases such that considering in addition the second highest percentage would provide more reliable coloring. Table 3.2 shows some example colored classes. The second row in Table 3.2 shows three classes 1, 2, and 3. For example class 2 occurs when the majority is `DT, T` and minority is `Stack`.

Table 3.2: Color classes for $iCFG$

Color Classes	Majority	Minority
1/2/3	DT, T	ArLo/Stack/CaLe
4/5/6	DT, ArLo	T/CaLe/Stack
7/8/9	DT, CaLe	ArLo/Stack/T
10/11/12	DT, Stack	T/CaLe/ArLo
13/14/15	T, ArLo	DT/CaLe/Stack
16/17/18	T, CaLe	DT/ArLo/Stack
19/20/21	T, Stack	DT/ArLo/CaLe
22/23/24	ArLo, Stack	T/DT/CaLe
25/26/27	ArLo, CaLe	Stack/DT/T
28/29/30	Stack, Cale	T/DT/ArLo

As an example, by applying the color classes in Table 3.2 to Figure 3.1(a), we can obtain the $iCFG$ shown in Figure 3.3(a). This $iCFG$ involves five color classes: 22, 4, 3, 10, and 1. From Table 3.2, we can see that a majority of those classes belong to: `ArLo-Stack, DT-ArLo, DT-T, DT-Stack, DT-T`. This is reasonable since the main functionality of the bubble sort algorithm is manipulating values in an array

and consequently the main action is the transfer of values from one location to another, which explains the large number of DT instructions. As demonstrated by the example, by using this extended control flow graph $iCFG$, we can capture more semantic information that might be helpful in identifying functions in binary code. Nonetheless, the $iCFG$ only contains control information about basic blocks, and it lacks other useful semantics, such as the way registers are manipulated and the way functions interact with each other. Hence, we introduce two other building blocks in addition to $iCFG$.

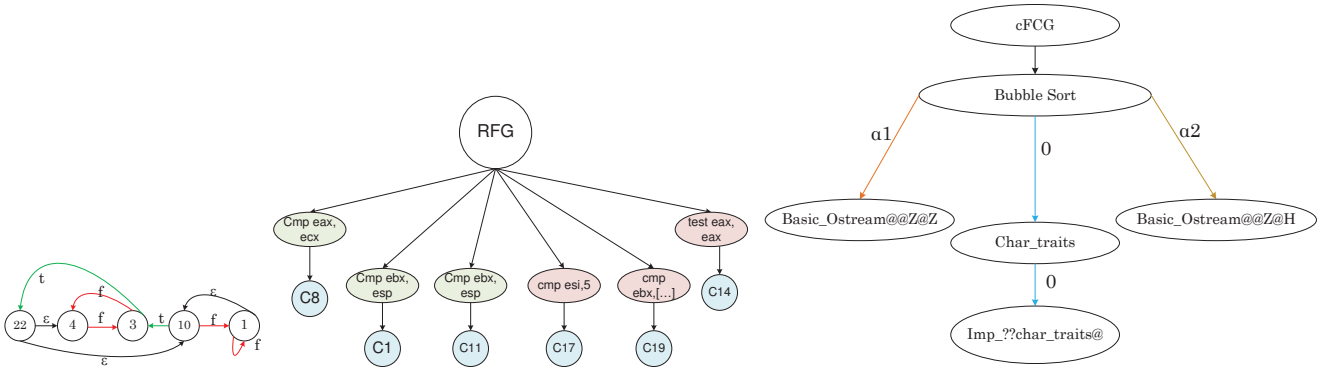


Figure 3.3: Enhanced classical representations for bubble sort function: (a) iCFG for bubble sort function (b) mRFG for bubble sort function (c) Function Call Graph for bubble sort function

B. Merged Register Flow Graph

As mentioned in Section 3.2, RFG is a binary code representation for capturing program behaviors based on an important semantics of the code, i.e., how registers are manipulated. The original RFG is designed for authorship attribution purposes, therefore it lacks support for some cases that are important for function identification: i) when both

operands of `cmp` are constants `(C)`, ii) when one of the operands is a constant and the other is a register (reg), iii) when both operands are memory locations `(ML)`, iv) when one of the operands is a memory location and the other is a register, and v) when the operands are a mixture of constants and memory locations. These cases are especially important for identifying functions in binary code, and hence we extend the RFG by adding several new classes as shown in Table 3.3.

Table 3.3: Updated classes of register access

Class	Arithmetic	Logical	Generic	Stack	C C	C Reg	ML ML	ML Reg	ML C
1	1	0	0	0	0	0	0	0	0
2	0	1	0	0	0	0	0	0	0
3	0	0	1	0	0	0	0	0	0
4	0	0	0	1	0	0	0	0	0
5	1	1	0	0	0	0	0	0	0
6	1	0	1	0	0	0	0	0	0
7	1	0	0	1	0	0	0	0	0
8	1	1	1	0	0	0	0	0	0
9	1	1	0	1	0	0	0	0	0
10	1	0	1	0	0	0	0	0	0
11	1	1	1	1	0	0	0	0	0
12	0	1	1	0	0	0	0	0	0
13	0	1	0	1	0	0	0	0	0
14	0	0	1	1	0	0	0	0	0
15	0	1	1	1	0	0	0	0	0
16	0	0	0	0	1	0	0	0	0
17	0	0	0	0	0	1	0	0	0
18	0	0	0	0	0	0	1	0	0
19	0	0	0	0	0	0	0	1	0
20	0	0	0	0	0	0	0	0	1

Moreover, as another improvement over the original RFG representation, we merge certain nodes inside an RFG, e.g., class one and class two together are equivalent to class five. In this manner, we can reduce the number of nodes to improve the efficiency in analyzing an RFG. Finally, since the original RFG depends only on the `cmp` instructions,

we also extend RFG instructions to the `test` instruction. After applying those extensions and modifications, we obtain the new representation $mRFG$, as shown in Figure 3.3(b). The $mRFG$ has three more nodes than its corresponding RFG; one of these is `test` instruction, and the other two are related to the immediate memory address and the constant. Moreover, we have merged the original classes (nodes in green): $C8 - C3$ to $C8$, $C1 - C1$ to $C1$, and $C5 - C14$ to $C11$. The reference to the new classes $C17$ and $C19$ may provide useful semantics about the functions, e.g., a bubble sort function implies it mainly deals with constants and sorts them in memory locations.

C. Color Function Call Graph

As mentioned in Section 3.2, traditional FCGs represent system calls in a binary code. Among a set of system calls $C = C_1, C_2, \ldots, C_n$, each call may be either local or external. To distinguish these, we extend FCGs with a color scheme as follows. The label function of labeling edges defines the label class α in two cases. For a local call, we only need one label, because local system calls are mostly related to compiler functions rather than to user functions. As to external calls, we define the label classes using a range of values $0 < \alpha < 1$, because we may have various external system calls potentially connecting to API that is very important for identifying functions. More precisely, we extend FCGs to a new representation which we call $cFCG$, using the label function defined as follows.

$$f(l) = \begin{cases} \alpha = 0 & \text{if } l \text{ is local system call} \\ \\ 0 < \alpha < 1 & \text{if } l \text{ is external system call} \end{cases}$$

As an example, having applied this new representation to our running example, we obtain the $cFCG$ shown in Figure 3.3(c). Besides serving as a building block of our proposed approach, the $cFCG$ representation may also be helpful in other related tasks by highlighting the difference between various types of calls, such as in malware classification through clustering external system calls, and for writing pattern-based signatures by providing analysts with a list of API call graph properties derived from external calls.

3.3.3 SIG: Semantic Integrated Graph

The building blocks introduced in the previous section provide complementary views on binary code by emphasizing different aspects of the underlying function semantics. Inspired by the work introduced in [126], in which different representations of source code are combined for vulnerability detection in source code (which is a different problem from ours as binary code lacks much of the useful information available in source code), we combine those different but complementary representations of binary code into a joint data structure in order to facilitate more efficient graph matching between different binary codes for identifying reused functions. Formally, a semantic integrated graph (SIG) is defined as follows.

Definition 3. A semantic graph $G = (N,V,\zeta,\gamma,\vartheta,\lambda,\omega)$ is a directed attributed graph where N is a set of nodes, $V \subseteq (N \times N)$ is a set of edges and ζ is edge labeling function which assigns a label to each edge: $\zeta \longrightarrow \gamma$, where γ is a set of labels. ϑ is a coloring function which colors each node $n \epsilon N$ based on statistical classes function λ. Finally, ω is a function for coloring $mRFG$.

Figure 3.4 shows a simple example of SIG with four nodes. Note that a SIG is a multigraph so two nodes may be connected by multiple edges, e.g., edges corresponding to $mRFG$ or $cFCG$. Moreover, `A,B,C,D` represent the outcomes of labeling nodes of function ϑ, and `a, b, c` are the outcomes of function ζ. C1, C5, C2, and C11 are outcomes of function ω. In the figure, the number in an oval shape represents the number of outcomes of the color function for $cFCG$, where 0 represents a local call, and α_1 and α_2 represent two different external calls.

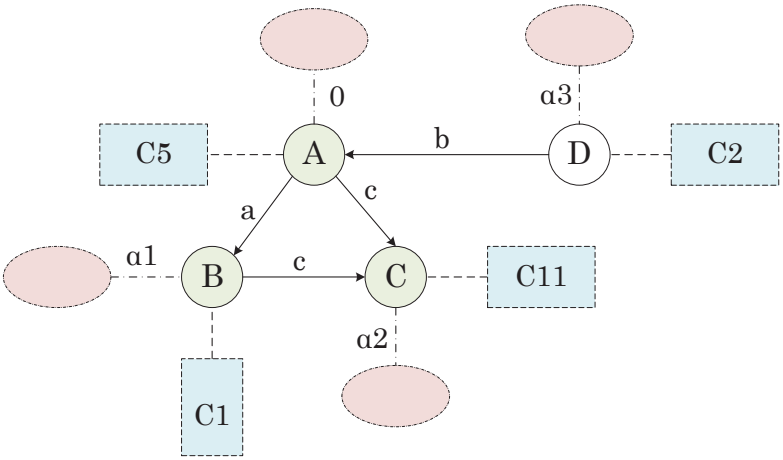


Figure 3.4: Simple example of SIG

To utilize the SIG for inexact matching and matching for fragments of function, we

need to consider meaningful subgraphs of SIG. Again inspired by [126], we decompose a SIG into short paths called $traces$, where each trace is a function $\varrho : S(N) \longrightarrow S(N')$ that maps a set of nodes in an SIG to another set of nodes according to given criteria, where $S(N)$ denotes the power set of N. The main advantage of such a definition is the composition of multiple traces always yields another trace, i.e., ϱ_0 and ϱ_1 can be chained together to $\varrho_0 \circ \varrho_1$. We define a number of elementary traces that serve as a basis for the construction of other traces, and some examples are shown in the following (each trace function also has other simpler forms, which are omitted due to space limitations).

$Out_{I,L,K}(Y) = \bigcup_{n \epsilon Y} \{m : (n,m) \epsilon V$ and $\vartheta(n,m) = I$ and $\lambda(n,m) = L$ and $\omega(n,m) = K\}$

$IN_{I,L,K}(Y) = \bigcup_{n \epsilon Y} \{m : (n,m) \epsilon V$ and $\vartheta(n,m) = I$ and $\lambda(n,m) = L$ and $\omega(n,m) = K\}$

$OR(\varrho_1, \varrho_2, ..., \varrho_n) = \varrho_1 \cup \varrho_2 \cup ... \cup \varrho_n$

$AND(\varrho_1, \varrho_2, ..., \varrho_n) = \varrho_1 \cap \varrho_2 \cap ... \cap \varrho_n$

The trace $Out_{I,L,K}$ returns all nodes reachable over edge I and connected to the node of the other graph with label L, all nodes connected with the node of the other graph with label K. Trace $IN_{I,L,K}$ is similarly defined to move backwards in the graph, and the

two traces `OR` and `AND` aggregate the outputs of other traces.

Example: SIG **for Bubble Sort Function.** Here, we give an example of SIG for the bubble sort function shown in Figure 3.5.

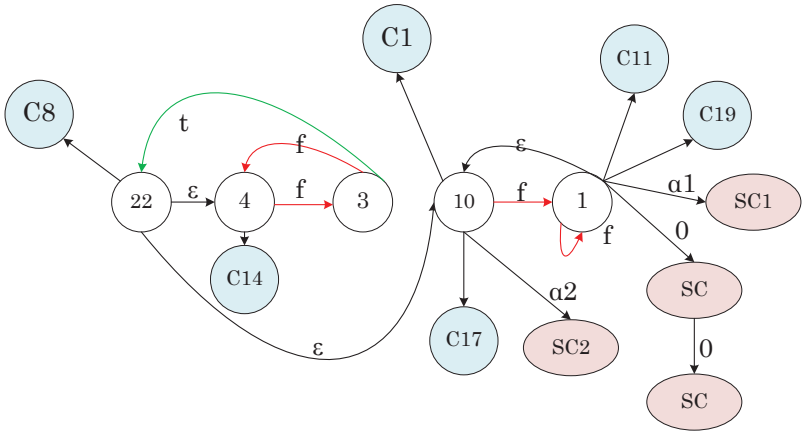


Figure 3.5: SIG for bubble sort function

As an example of SIG trace, we show the traces of nodes `22, 4, and 3` in Table 3.4. Also, we show just one example of OR trace and AND trace as well. Moreover, we extract additional features from the SIG shown in Figure 3.5, as depicted in Table 3.5. The features in Table 3.5 include total number of nodes, number of control edges(e.g., `22`), number of flow edges (e.g., `0`), number of flow nodes (e.g., `C8`), and etc. Those features together with the SIG traces are sufficient for exact matching of SIGs, and we will discuss inexact matching in next section.

Table 3.4: Part of traces for SGF bubble sort function

Node	Traces	Traces Type
22	ϵ, ϵ, C8 t	out in
4	f, C14 ϵ, f	out in
22 OR 4	ϵ, C8, C14, f f, t, ϵ	out in
4 AND 3	f f	out in

Table 3.5: Graph features for exact matching

Features	Frequency
Total # of Nodes	15
Total # of Edges	18
# of Control Nodes	5
# of Control Edges	8
# of Call Nodes	4
# of Call Edges	4
# of Register Nodes	6
# of Register Nodes	6

3.3.4 Graph Edit Distance

For inexact matching between SIGs, we will need a distance metric. In this chapter, we employ the graph edit distance for this purpose. The edit distance between two graphs measures their similarity in terms of the number of edits needed to transform one into the other [71]. We implement this concept as follows. Given two SIGs, we define the following two elementary traces to transform one graph into another: `Edge-edit traces`, including κ_r, re-labels the edge. `Node-edit traces`, including ν_r, re-colors the node by merging nodes from the other graph into one node. An edit edge $V_{G,H}$ between two SIGs `G` and `H` is defined as a set of sequences (ϱ_1, ϱ_2, ... , ϱ_n) of traces such that `G` = ϱ_n

(ϱ_1(ϱ_{n-1}(H)(... ϱ_1(H) ...)). To quantify this similarity, the weight of all edit traces is measured, i.e., $V = (\varrho_n, \varrho_2, ... , \varrho_n)$ as $w(V) = \sum_{i=1}^{n} w(\varrho_i)$. The edit distance between two SIGs is thus defined as the minimum weight of all edit edges and nodes between them, i.e., $sim(G, H) = min\ w(V_{G,H})$. The distance measure between the nodes follows the same reasoning, with operations instead of traces. In Algorithm 1, we calculate the graph edit distance between two SIGs, G and H, by measuring the cost of transforming G to H. The algorithm starts by labeling the edges of the two graphs as mentioned earlier, and then checks the cost of transforming each node in G to nodes in H, and finally calculates the total cost.

Algorithm 1: Graph Edit Distance

input : G: semantic integrated graph
H: semantic integrated graph
R: total set of edges
e: last element of edges
output: sim: Similarity result for two graphs

begin
`Steps 1 and 2 for edge labeling`
1. $(G, V') \leftarrow ExtractEdges\,(G, V)$;
2. $(H, V") \leftarrow ExtractEdges\,(H, V)$;
for *each* v_i *in* V' **do**
  for *each* v_j *in* $V"$ **do**
    if $v_j.\varrho$ = $v_i.\varrho$ **then**
      $sim(G, H) \leftarrow \varrho$ in v_j;
for *each* r *in* R **do**
  for *each* $1 <$ *e* **do**
    $w(V) \leftarrow w\,(\varrho_i)$;
return $sim(G, H) = minw(V_{G,H})$

We define the dissimilarity between two SIGs G and H as follows:

Definition 5. The dissimilarity $\rho(G, H)$ between two SIGs is a value in $[0, 1]$, where

0 indicates the graphs are the most similar and 1 the least similar, as formulated in the following.

$$\rho(G,H) = \frac{w(V_{G,H})}{|N_G| + |N_H| + |V_G| + |V_H| + |\varrho_G| + |\varrho_H|}$$

Where $w(V_{G,H})$ is the weight cost of traces, $|N_G|$ the number of nodes in G, $|N_H|$ the number of nodes in H, $|V_G|$ the number of edges in G, $|V_H|$ the number of edges in H, and $|\varrho|$ the number of traces in both SIGs.

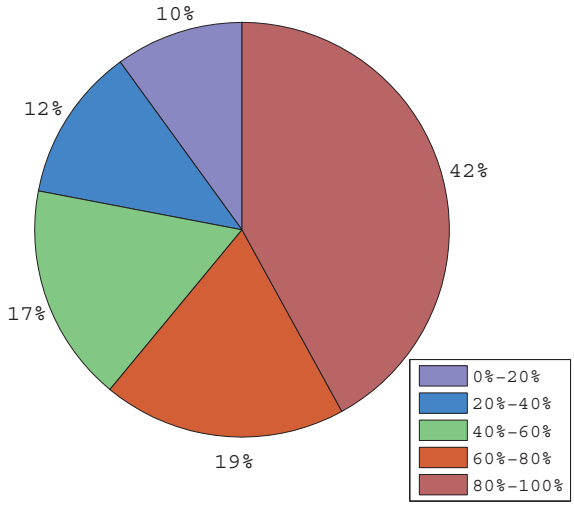
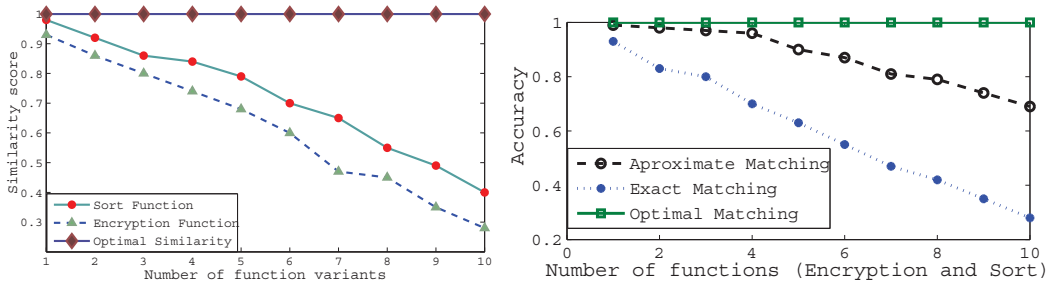


Figure 3.6: Similarity statistics of function variants

3.4 Experimental Results

We implement and test the proposed technique, *SIGMA*, with variants of sorting algorithms and encryption algorithms in order to evaluate the effectiveness and correctness

of the proposed method. We employed two variants for each sort function (e.g., bubble, quick, merge, and heap) and two for each encryption algorithm (e.g., RC4, MD5, Advanced Encryption Standard (AES), and the tiny encryption algorithm (TEA)). Using the proposed method, similar scores among these samples are calculated based on the graph edit distance and dissimilarity formulas introduced in previous section. The results are depicted in Figure 3.6.



(a) (b)

Figure 3.7: (a) Relation between the number of variants with the similarity score (b) Accuracy of using exact and approximate matching

According to Figure 3.6, the similarity score shows a promising value with about 80% similarity score pairs ranging from 0.42 to 1. Furthermore, the similarity score on pairs ranging from 0 to 0.2 is only about 10%. The results clearly show that our approach can capture common characteristics between functions relatively well. The occurrences of low-score pairs are mainly due to the significant differences in sizes of functions and variants, and also the number of nodes, edges, and traces may be observably different. For instance, the number of nodes in a bubble sort variant a is 15, whereas for variant b is 22; the number of edges in each one is 18 and 43, and the number of traces is 147 and 278, respectively. In Table 3.6 and Table 3.7, the similarity matrix shows similar scores of each

pair of encryption functions. The values (100%) in the main diagonal are the similarity scores for the variants when compared to themselves. In both tables, we represent RC4 by R, TEA by T, AES by A, MD5 by M, Bubble by B, Quick by Q, Heap by H, and Merge by M.

Table 3.6: Similarity between sort function variants

	B.1	**B.2**	**Q.1**	**Q.2**	**M.1**	**M.2**	**H.1**	**H.2**
B.1	100%	93%	71%	67%	62%	73%	65%	62%
B.2	96%	100%	79%	80%	70%	72%	60%	68%
Q.1	79%	83%	100%	94%	76%	71%	65%	60%
Q.2	71%	69%	95%	100%	79%	77%	74%	65%
M.1	67%	76%	66%	68%	100%	97%	70%	74%
M.2	73%	69%	77%	78%	94%	100%	70%	72%
H.1	69%	67%	74%	73%	79%	79%	100%	96%
H.2	72%	71%	64%	69%	79%	78%	95%	100%

Table 3.7: Similarity between encryption function variants

	R.1	**R.2**	**T.1**	**T.2**	**M.1**	**M.2**	**A.1**	**A.2**
R.1	100%	86%	68%	57%	52%	61%	57%	62%
R.2	89%	100%	74%	66%	53%	72%	50%	59%
T.1	72%	79%	100%	87%	66%	61%	55%	67%
T.2	68%	62%	89%	100%	72%	67%	69%	55%
M.1	57%	69%	58%	51%	100%	91%	78%	74%
M.2	63%	67%	67%	70%	92%	100%	78%	72%
A.1	69%	57%	64%	68%	79%	75%	100%	94%
A.2	62%	71%	69%	64%	70%	73%	89%	100%

We can see from Table 3.6 and Table 3.7 that similarity scores among the sort functions are higher than those among encryption functions. This is due to the fact that the steps of sorting are similar among different algorithms but the steps of encryption functions vary significantly with each algorithm.

In Table 3.6, the similarity between heap and other algorithms are lower, because the steps of heap sort are significantly different from the other sorting algorithms' steps.

Table 3.8: Dissimilarity between sort and encryption functions

	Bubble.1	**Quick.1**	**Merge.1**	**Heap.1**
RC4.1	86%	93%	79%	87%
TEA.1	96%	91%	79%	89%
MD5.1	79%	88%	90%	94%
AES.1	89%	91%	95%	84%

In Table 3.7, the similarity scores show that RC4 is more similar to TEA, than MD5 is to AES. This is due to the fact that RC4 and TEA have steps in common in the encryption process. In Table 3.8, we list the calculated dissimilarity scores between sorting algorithms and encryption algorithms.

3.5 Summary

The reverse engineering of binary code is an important but challenging task that demands automated techniques for preprosessing and cleaning the code. The identification of reused functions in binary code is one of the important aspect of this issue that has received limited attention in comparison with other aspects of binary analysis. In this chapter, we have presented a novel approach called *SIGMA* for effectively identifying reused functions in binary code. Instead of relying on one source of information, our approach combines multiple representations into one joint data structure SIG. *SIGMA* also supports inexact matching and exact matching based on traces of the SIG which deals with function fragments. Both experimental results and case study have demonstrated the effectiveness of our method, and we have described several potential improvements to the approach in the previous section.

Chapter 4

Identifying Free Open-Source Software Functions in Binary Code

4.1 Overview

Identifying free open-source software (FOSS) packages on binaries when the source code is unavailable is important for many security applications, such as malware detection, software infringement, authorship attribution, and digital forensics. This capability enhances both the accuracy and the efficiency of reverse engineering tasks by avoiding false correlations between irrelevant code bases. Although the FOSS package identification problem belongs to the field of software engineering, conventional approaches rely strongly on practical methods in data mining and database searching. However, various challenges

in the use of these methods prevent existing function identification approaches from being effective in the absence of source code. To make matters worse, the introduction of code transformation techniques, the use of different compilers and compilation settings, and software refactoring techniques has made the automated detection of FOSS packages increasingly difficult. With very few exceptions, the existing systems are not resilient to such techniques, and the exceptions are not sufficiently efficient.

To address this issue, this chapter proposes, *FOSSIL*, a novel resilient and efficient system that incorporates three components. The first component extracts the syntactical features of functions by considering opcode frequencies and applying a hidden Markov model statistical test. The second component applies a neighborhood hash graph kernel to random walks derived from control flow graphs, with the goal of extracting the semantics of the functions. The third component applies z-score to the normalized instructions to extract the behavior of instructions in a function. The components are integrated using a Bayesian network model which synthesizes the results to determine the FOSS function. The novel approach of combining these components using the Bayesian network has produced stronger resilience to code transformation methods. We evaluate our system on two datasets including real-world projects whose use of FOSS packages is known and malware binaries for which there are security and reverse engineering reports purporting to describe their use of FOSS. We demonstrate that our system is able to identify FOSS packages in real-world projects with a mean precision of 0.95 and with a mean recall of 0.85. Furthermore, *FOSSIL* is able to discover FOSS packages in malware binaries that

match those listed in security and reverse engineering reports.

4.2 Preliminaries

In this section, we show the challenges faced in automating the process of identifying FOSS functions in malware binaries. Second, we briefly describe the threat model and highlight the in-scope and out-of-scope threats of this work. We then provide an overview of our system. Finally, we present our criteria in selecting FOSS packages for evaluation.

4.2.1 Challenges

In automating the process of identifying FOSS functions in malware binaries, several challenges are typically encountered. The first is *usability*. Immediate insights obtained about a binary file from a system to highlight FOSS packages will give reverse engineers a direction to start their investigations. The existing approaches for the purpose of binary search engine, clone detection, or function identification return the top-ranked candidate functions, while these results are helpful if the repository contains a function that exhibits a high degree of similarity to the target function. Moreover, because of the effect of different compilers, compiler optimization, and obfuscation techniques, a given unknown function is less likely to be very similar to the right function in the repository, and there is little advantage in returning a list of matches with low degrees of similarity. A resilient system should be able to identify the matched pairs with a controller process that can synthesize the available knowledge. The second challenge is *efficiency*. An efficient system

can help reverse engineers to find matches on the fly. To efficiently extract, index, and match features from program binaries in order to detect a given target function within a reasonable time, considering the fact that many known matching approaches imply a high complexity is challenging. The third challenge is *robustness*. The distortion of features in the binary file may be attributed to different sources arising from the platform, the compiler, or the programming language, which may change the structures, syntax, or sequences of features. Hence, it is challenging to extract robust features that would be less affected by different compilers, slight changes in the source code as well as obfuscation techniques. The fourth challenge is *scalability*. Reverse engineers deal with large numbers of binaries on a daily basis, so it is necessary to design a system that could scale up millions of binary functions. Accordingly, it is important to consider the factors that may degrade the performance of FOSS package identification as the repository size increases. The fifth challenge is *stability*. One of the most important concerns in the design of a system is to provide a component to update the repository, when a new version of a FOSS package is released. The update process should be supported by a system that does not need to re-index the whole package.

4.2.2 Threat Model

Our system is designed to assist, instead of replace, reverse engineers in various use cases, such as digital forensic analysis (e.g., clustering a group of functions based on similar fingerprints) or software vulnerability disclosure (e.g., linking the code fragment of a

binary to a known vulnerable/buggy function). In what follows, we further clarify the threat model and scope of this thesis.

In-Scope Threats. In designing the features and methodology of our system, we have taken into consideration certain potential threats. First, adversaries may intentionally apply code transformation techniques to alter the syntax of binary files. Second, since the syntax of a program binary can be significantly altered by simply changing the compilers or compilation settings, adversaries may make such changes to evade detection. Finally, adversaries may reuse FOSS packages through modifying and adapting them to intentionally avoid detection by our system. We show how our system resists and survives these threats in Section 4.4.7. Furthermore, we can certainly envision many countermeasures taken by future malware writers to evade detection by our system, and hardening our system against such countermeasures will be an ongoing process.

Out-of-Scope Threats. As previously mentioned, our system is not intended to completely replace reverse engineers. Thus, the focus of our system is not on general reverse engineering tasks, such as unpacking and de-obfuscating binaries (although we later discuss how our system handles some obfuscation methods intended to evade detection by our system), but rather on discovering user functions. Our system assumes the binary is already de-obfuscated. In addition, cases where the code is encrypted in order to reduce code size or to prevent reverse engineering are also out of the scope of our system.

4.2.3 System Overview

The main analytical process is divided into four stages, as shown in Figure 4.1.

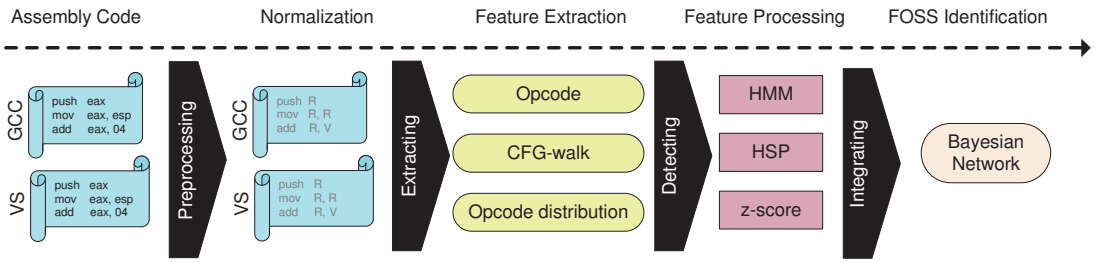


Figure 4.1: Overview of the proposed system

A preprocessing stage prepares normalized disassembled instructions, followed by feature extraction once the FOSS packages are collected. Then, different detection methods are applied to the extracted features, and the repository is explored for the purpose of identification. Further, the results of these detection methods are integrated using a Bayesian network model, making it possible to identify FOSS functions and label them in the binary. In the first step, the assembly instructions are normalized. The second step extracts opcodes, CFG-walks, and opcode frequency distribution features (Section 4.3.1). A Bayesian network controls the application of different detection methods, including HMM, HSP, and z-score, to the extracted features (Section 4.3.3). More specifically, the input of first method (HMM) is function opcodes, which are normalized according to the function length. A hidden Markov model (HMM) is applied to these opcode frequencies, as it can efficiently detect the behavior of a function (Section 4.3.3). The second method,

HSP, accepts control flow graph walks, which are labelled by applying the kernel function for each node together with its neighbors efficiently, as described in Section 4.3.3. Depending on the output of this component, either the function is identified, or the third component of the model is checked. To achieve this, the opcode frequencies are used as input and are converted into a probability function whose characteristics are analysed with the use of z-score, as described in Section 4.3.3. These statistical features usually capture the relationship between instructions and the behavior of the function. Finally, as described in Section 4.4, we evaluate our approach in terms of efficiency against a set of FOSS packages compiled with different compilers and compilation settings as well as in terms of robustness against code transformation techniques.

4.2.4 FOSS Packages

Collecting FOSS packages is a crucial step in evaluating our system. To build a repository of FOSS functions, the packages are chosen using statistics that show the prevalence of FOSS libraries [20], studies of malware behavior [49, 68, 127, 129], and technical reports [38, 85, 97]. We either collect the executable files or compile these packages according to their dependencies. We tailor our system to C-family compilers because of their popularity and widespread use, especially in the development of malicious programs [88]. The FOSS packages were created to perform various functionalities as partially listed in Table 4.1.

Table 4.1: Example of FOSS packages

Functionality	
Compression (e.g., info-zip)	MSDN libraries (e.g., NSPR)
Database management (e.g., SQLite)	Network operations (e.g., webhp)
Encryption (e.g., TrueCrypt)	Random number generation (e.g., Mersenne Twister)
File manipulation (e.g., libjsoncpp)	Secure connection (e.g., libssh2)
Hashing (e.g., Hashdeep)	Secure protocol (e.g., openssl)
Image compression (e.g., openjpeg)	Terminal emulation (e.g., xterm)
Multimedia protocols (e.g., Libavutil)	XML parser (e.g., TinyXML)

4.3 Design and Implementation of Our System

In this section, we introduce our system design and describe the features in detail. We also provide an overview of the implementation environment.

4.3.1 Features

In what follows, we introduce opcodes, CFG-walks, and opcode frequency distribution features used in our system.

Opcodes. Opcodes are defined as operational codes, which can be used to efficiently detect obfuscated or metamorphic malware [40]. However, Bilar et al. show that prevalent opcodes (e.g., `mov`, `push`, and `call`) do not make good indicators of malware samples [40], and based on such opcode frequencies, the resultant degree of similarity between two files could potentially be marred [55]. Therefore, we propose a way to avoid this phenomenon and to give each opcode its actual relevance by applying feature ranking based on mutual information [105] in order to consider only the top-ranked opcodes.

Control Flow Graph Walks. Control Flow Graphs (CFGs) consist of a set of basic

blocks, each of which represents a sequence of instructions without an intervening control transfer instruction. In the literature, CFGs have been used to detect variants of malware [46]. Exact matching of the CFG itself does not offer much help towards our goal, since the CFG might change, for instance, due to the effect of compilers and optimization settings. Consequently, we decompose a CFG into a set of walks, taking into consideration the interactions within these walks. By doing so, we will be able to convert CFGs into a set of semantic relations (walk interactions) such that when a malware uses part of a FOSS function to implement a specific functionality, it would be captured based on these semantic relations. In the literature, the random walk kernel [69] and the shortest path kernel [122] are amongst the most prominent graph kernels that have been used. A graph is decomposed into sequences of nodes generated by walks; it counts the number of identical walks that can be found in two graphs. We propose an instance of the substructure fingerprint kernel suitability for the analysis of CFG walk relations.

Example: Suppose a CFG consisting of ten basic blocks ($BB_0, \cdots, BB_9$) as shown in Figure 4.2a. To illustrate the random walk selection, we consider two nodes BB_0 and BB_7, where the path between them (BB_0,BB_4,BB_6,BB_7) with a distance of 3 is highlighted in Figure 4.2b. To reduce time complexity, we consider a radius for our random walk, which is the shortest path with neighboring nodes. In our experiments, we consider radius = $\{0, 1, 2\}$, as illustrated in Figures 4.2b-d, respectively. By choosing the radius equal to 0, the information is only about node BB_0 and node BB_7 as shown in Figure 4.2b. When $r = 1$ the (BB_0, BB_4), and (BB_6, BB_7) pairs represent the structural information

depicted in Figure 4.2c. The walks when radius is equal to 2 are (BB_0, BB_4, BB_6), and (BB_4, BB_6, BB_7). Through our experiments, we find that a radius of 2 is the best choice in terms of efficiency.

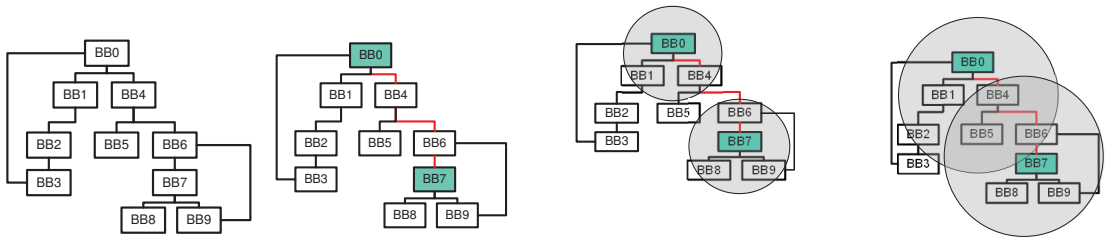


Figure 4.2: Example of random walks between two nodes BB_0 and BB_7 in (a) CFG of a function, by considering three radius (r) values: (b) $r = 0$, (c) $r = 1$, and (d) $r = 2$

Opcode Frequency Distribution. We select the opcode frequency distribution feature based on the following observations obtained from our experiments. We first consider the simple hypothesis that FOSS functions performing the same task usually exhibit similar distributions of opcodes [40]. Second, considering the fact that the area under a frequency distribution curve is always 1, we calculate the percentage of top-ranked opcode frequencies under the distribution. Third, the distribution of various opcodes conforms to a consistent distribution shape [51] when it is related to a specific FOSS function, even if the function is modified; since the semantics will be preserved [75] and may be discovered by the distribution.

4.3.2 Feature Selection

We are interested in extracting features that represent the functionality and semantics of binary functions. We extract different representations of code properties, but only a subset of these representations may serve as indicators of the semantics of a function. Hence, we aim to select features that best preserve the semantics of a function. As such, instead of relying only on syntax-based features obtained from feature templates [63], we propose capturing different function features at various abstraction levels of the binary code. Furthermore, we consider how to efficiently extract features from binaries and how to efficiently store them in a repository.

Opcode ranking. The first category of our features are opcodes. By applying mutual information-based ranking [45] to the opcodes and corresponding functions of each FOSS family, we reduce the feature set size to effectively represent the properties of coding functionality. The opcodes with highest ranking values will be used to calculate opcode frequency distributions and to color CFG-walks. We employ mutual information to determine the degree of statistical dependence of two variables X and Y as follows:

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

where x is the opcode frequency, y is the class of FOSS function (e.g. `sqlite3MemMalloc`), $p(x)$ and $p(y)$ are the marginal probability distribution of each random variable, and $p(x,y)$ is the joint probability of X and Y [113]. The joint and

marginal distributions are computed over the number of function variants N (For each function we have different versions such as when it is compiled with VS or GCC). These distributions are computed between class (function label) and feature as follows [113]:

$$P(x) = \frac{1}{N}\sum_{i=1}^{N} \ell_{[x_i=x]}, P(y) = \frac{1}{N}\sum_{i=1}^{N} \ell_{[y_i=y]}, P(x,y) = \frac{1}{N}\sum_{i=1}^{N} \ell_{[x_i=x \bigwedge y_i=y]}$$

Graph coloring. Relying on structural information to identify functions which are semantically similar is not sufficient given the fact that two distinct functions may still have identical CFGs [29]. This shortcoming is addressed by the idea of graph coloring, where the content of each basic block is also taken into consideration. We use the graph coloring technique proposed in [84] to color the nodes based on the group of instructions in a basic block. This technique categorizes the instructions according to their semantics; for instance, `push` and `pop` opcodes are classified in one class (e.g., Stack operation). As a result, there are fewer possibilities for an attacker to find semantically equivalent instructions from different classes. Furthermore, the possible variations in coloring that can be generated with instructions from different classes are much fewer than the possible variations on the instruction level [84]. We apply the coloring technique on the normalized instructions (including the opcodes and operands) by considering only the top-ranked opcodes. Finally, we assign a weight to each edge by aggregating the colors of the source and destination nodes. For instance, if there is an edge between node A to node B, and they are colored in 2 and 8 respectively, the weight of this edge would be 10.

Opcode importance. We follow the model used in [40] to measure the importance of opcodes. The top-ranked opcodes are further processed through converting the frequencies into a histogram and measuring the area of intersection based on the probability distribution. This step illustrates the importance of each ranked opcode in terms of function behavior. The most important opcodes will be used by the third component.

4.3.3 Detection Method

In this section, we introduce the components of our detection system: the hidden Markov model, the neighborhood hash graph kernel, and the z-score, where the Bayesian network integrates these components.

A. Hidden Markov Model

After the mutual information between the FOSS function and the individual opcodes is computed, an opcode relevance file based on the top-ranked features for each function is created. These top-ranked opcodes are used for the hidden Markov model (HMM) with chi-squared testing. Thus, the functions are scored (according to the opcode sequences) and classified based on whether they belong to the FOSS or not. Following this, we apply chi-squared distance with a HMM, as a way to create a confidence interval for this component. HMM was picked to be the initial component since it is computationally efficient [121].

In the HMM model, the states represent the sequence of instructions, however, they

are not fully observed; yet, the hidden states can be estimated by observing the sequences of data [120]. To discover the hidden states (e.g., instructions related to inline functions), we apply data flow analysis such as read and write dependencies as what follows. There exists a data flow dependency between two instructions i_1 and i_2 according to the following rules: (i) i_1 reads from a register or a memory address, and i_2 writes to the same register or memory address. (ii) i_1 writes to a register or memory address and, i_2 writes to the same register or memory. (iii) i_1 writes to a register or memory address, and i_2 reads from the same register or memory. Consequently, if an instruction (or set of commands) shows no evidence of a data flow dependency, it is tagged as a hidden state. It should be noted that "instruction side effect" (which flag is manipulated) are treated as observations. Therefore, such observations will be annotated to the states.

In what follows, we describe the chi-squared distance, which is combined with the HMM. The main objective is to determine the preeminent characteristics of the probability distribution of statistical opcode variable Z. The best way to find out which hypothesis is the best match for an observed sequence of samples $(Z_1, Z_2, \ldots, Z_n)$ is to use statistical testing. In fact, the Pearson's χ^2 statistic [65] is widely employed to confirm whether or not the discrepancies between the observed and expected data are significant. We denote this test as T^2, which is given by [120, 121]:

$$T^2 = \sum_{(i=1)}^{z} \frac{(\hat{m}_i - m_i)^2}{m_i} \leq \chi^2(\alpha, v-1) \tag{4.1}$$

where $\hat{m}_i$ and m_i are the normalized frequencies of opcodes in the testing phase and

training phase, respectively; $(\alpha, v-1)$ represent type I error rate, and the degrees of freedom, respectively. Finally, based on the comparison results of T^2 and $\chi^2(\alpha, v-1)$, the decision threshold is acquired. For more details, we refer the reader to [120, 121].

B. Neighborhood Hash Graph Kernel

As Gartner et al. [69] show, the distance between CFG-walks (node-node interaction relations) has a crucial impact on obtaining the semantics of a graph. To be more precise and to take into consideration the subgraph pairs (not only pairs), we apply a hash subgraph pairwise (HSP) [132] kernel based method to represent the structural information of the node interactions in a linear time using hierarchical hash labels.

The label pair feature space of graph H for each label pair (l_i, l_j) is defined as follows [132]:

$$\varphi_{l_i,l_j}(H) = \sum_{q=0}^{\infty} \tau_q |\{w \in W_q(H) : f_1(w) = l_i \wedge f_{q+1}(w) = l_j\}| \tag{4.2}$$

where $W_q(H)$ is the set of all possible walks with q edges in graph H, $f_1(w)$ is the first node of walk w, $f_{q+1}(w)$ is the last node of walk w, and $(\tau_0, \tau_1, \cdots, \tau_q)$ are weights of edges. Each edge is weighted by the summation of source and destination node colors (e.g., $c(i) + c(i+1)$, where function c calculates the node color, discussed in Section 4.3.2). In correspondence with feature map provided above, the graph kernel function

based on label pairs is calculated as in [132] as follows:

$$K(H,H^{'}) = \langle \varphi(H), \varphi(H^{'}) \rangle = \langle L(\sum_{i=0}^{\infty} \tau_i E^i) L^T, L^{'}(\sum_{j=0}^{\infty} \tau_j E^j) L^{'T} \rangle$$

$$= \sum_{m=0}^{|k|} \sum_{n=0}^{|k|} \left[L \left(\sum_{i=0}^{\infty} \tau_i E^i \right) L^T \right]_{mn} \left[L^{'} \left(\sum_{j=0}^{\infty} \tau_j E^j \right) L^{'T} \right]_{mn} \quad (4.3)$$

where E^i is the adjacency matrix of H, and L is the labeled matrix of H.

The manner by which the label process is made is described as follows. We denote a label as a binary vector $e = \{u_1, u_2, \ldots, u_r\}$ consisting of r-bits (0 or 1), representing the presence (1) of the group of instructions (discussed in Section 4.3.2) in a node. Let $XOR(e_i, e_j) = e_i \oplus e_j$ symbolise the XOR operation between two bit vectors of e_i and e_j. Let $ROT_o(e) = \{u_{o+1}, u_{o+2}, \ldots, u_r, u_1, \ldots, u_o\}$ denote the rotation (ROT_o) operation for $e = \{u_1, u_2, \ldots, u_r\}$, which shifts the last $r - o$ bits to the left by o bits and moves the first o bits to the right.

In order to compute the neighborhood hash of a graph, we first obtain the set of adjacent nodes $N^{adj}(n) = \{N_1^{adj}, \ldots, N_d^{adj}\}$ for each node n, and then calculate a neighborhood subgraph hash label for every node, using the following equation 4.4 [133], where $l_i(n)$ indicates bit label of node n.

$$l_{i+1}(n) = NH(n) = ROT_1(l_i(n)) \oplus (ROT_o)(l_i(N_1^{adj})) \oplus \cdots \oplus ROT_o(l_i(N_d^{adj})) \quad (4.4)$$

As a way to differentiate between an outgoing and an ingoing edge, we set two ROT_o operations. If the edge $n_1 n$ is an incoming edge to node n, let $ROT_o = ROT_2$; if the edge $n n_1$ is an outgoing edge of node n, let $ROT_o = ROT_3$. It is worth nothing that

$l_0(n)$ describes the information of node n, while $l_1(n)$ represents the label distribution of node n and its adjacent nodes. Finally, the structural information of subgraph of radius i is presented by $l_i(n)$. According to our experiments, we find a radius of $r = 2$ is the best choice for our system.

According to hierarchical hash labels, the graph kernel is defined as [132]:

$$K(H, H^{'}) = \sum_{r=0}^{r^*} \beta r \left\langle Lr \sum_{i=0}^{\infty} \left(\tau_i E^i \right) (Lr)^T, Lr^{'} \sum_{j=0}^{\infty} \left(\tau_j E^j \right) (Lr)^{'T} \right\rangle$$

$$= \sum_{r=0}^{r^*} \sum_{m=0}^{|k|} \sum_{n=0}^{|k|} \beta r \left[Lr \left(\sum_{i=0}^{\infty} \tau_i E^i \right) Lr^T \right]_{mn} \left[Lr^{'} \left(\sum_{j=0}^{\infty} \tau_j E^j \right) Lr^{'T} \right]_{mn} \tag{4.5}$$

where E is the adjacency matrix of H and $L_0, L_1, \ldots, L_r$ are the hierarchical hash labels of H. For more details, we refer the reader to [132, 133].

From a practical perspective, the whole process involved in calculating hierarchical hash labels is linear with respect to the size of the graph [132, 133]. Consequently, computing the similarity between two control flow graphs will be equivalent to comparing the set of hash values.

C. Calculation of Z-score

The last component concerns the distribution of opcode frequencies, since each set of opcodes that belong to a specific function will likely follow a specific distribution due to the functionality they implement. For this purpose, we utilize the z-score in order to convert these distributions into scores. In essence, a z-score $Z = \frac{(x-\mu)}{SD}$ indicates how many standard deviations an element is from the mean.

We calculate the z-score for each opcode distribution to facilitate accurate comparisons. Based on the possible values for the z-score, we obtain a curve distribution, where the area under the curve provides one feature value for each function. The area under the curve is calculated as $P(min < Z < max) = P(Z < max) - P(Z > min)$.

D. Bayesian Network Model

We use a Bayesian Network (BN) model for measuring the knowledge obtained from each component and for automating the interaction amongst these components. In addition, BN can depict the relations between the three proposed components, and would encode probabilistic relationships among their outputs. Moreover, situations where certain data for such components are not sufficient for identification can be handled by BN. A BN can be used to gain knowledge from a FOSS function identification problem domain as well as to predict the consequences of intervention, since it can be used to learn causal relationships. This feature is very important in the case of modifications performed by malware writers. Hence, the BN can capture both causal and probabilistic relationships, and is an ideal representation for combining prior knowledge and data.

The joint probability function would be calculated as $P(f, H, W, Z) = P(f|H, W, Z)P(f|H, W)P(f|H)P(H)$, and the probability is defined with Bayes' law by equation 4.6:

$$p(y|\vec{x}) = \frac{p(y)p(\vec{x}|y)}{p(\vec{x})} \tag{4.6}$$

where *p ($\vec{x}|y$)* is the probability of a possible input $x = (x_1, \ldots, x_n) \in \Upsilon^n$ given the output $y = (y_1, \ldots, y_n) \in \Gamma^n$. We define a set of conditional probabilities (factors) Ψ_1, Ψ_2, and Ψ_3 for our three components. Through extensive experiments applying logistic regression [50], we found that, for our experimental settings, the best values for these factors are 0.45, 0.35, and 0.2, respectively. These factors are based on all possible features in the components, and thus represent more explicitly the underlying probability distribution of the features in each component. Each part of the joint probability is obtained by the equation 4.7.

$$p(y|x) = \frac{p(x,y)}{p(x)} = \frac{p(x,y)}{\sum\limits_{y} p(x,y)} = \frac{\frac{1}{Z}\Pi_{s\in S}\Psi_s(x,y)}{\frac{1}{Z}\sum\limits_{y'}\Pi_{s\in S}\Psi_s(x_s,y_s)} = \frac{1}{Z(x)}\Pi_{s\in S}\Psi_s(x,y) \quad (4.7)$$

As previously described, our system encompasses three main components to identify a FOSS function f: HMM (H), CFG-walks (W), and z-score (Z). Each component provides particular knowledge about the FOSS function, and the provided knowledge is measured by a factor Ψ_s. If the factor $\Psi_s \leq \Omega$, where Ω is a probability threshold value set by the Bayesian network, then our system is automatically transferred to another component, which means that the knowledge obtained from the current component is not sufficient. In addition, each component has a direct effect on the use of the other one; for instance, component H has a direct effect on component W. The situation can thus be modeled with a Bayesian network model.

where Ψ_s represents the factor of the component $s = \{1, 2, 3\}$; $\prod$ represents the summation of the product of probabilities from each component; Z is the probability distribution [92]; x is the set of features in each component; and y is the set of functions. We therefore obtain the following equation: $p(y, \vec{x}) = p(y)\frac{1}{Z(\vec{x})}\Pi_{s \in S}\Psi_s(\vec{x}, y)$.

4.4 Evaluation

In this section, in order to evaluate the effectiveness of our system, we test 160 real projects that reuse FOSS packages. The performance criterion is the accuracy (F_2 measure) with which our system identifies the FOSS functions in malware binaries. In addition, we examines the effect of code transformation on the proposed system.

4.4.1 Dataset Preparation

We manually gather a collection of FOSS packages from different sources and store them along with their features in a repository. Developing this repository is the first step towards the ultimate goal of building a large index of FOSS packages. To determine which FOSS packages are most widely incorporated, it is helpful to study code reuse on open repositories such as Github. The method for selecting the reused code involves assessing both the most popularly projects and the most reused libraries in modern malware based on the existing reports. After gathering the list of FOSS packages, we download their source code and compile them with Visual Studio (VS) 2010, VS 2012, and GNU Compiler Collection (GCC) 5.1 compilers. Furthermore, we obtain binaries and their PDBs

Table 4.2: Excerpt of the selected FOSS packages

Project	Version	No. Fun.	Size(kb)	Project	Version	No. Fun.	Size(kb)
7zip/7z	15.14	133	1074	lshw	B.02.18	2090	2545
7zip/7z	15.11	133	1068	lzip	1.19	3341	1552
avgntopensslx	14.0.0.4576	3687	976	Mersenne Twister	1.10	321	2608
bzip2	1.0.5	63	40.0	miniz	2.8	327	121
expat	0.0.0.0	357	140	ncat	0.10rc3	462	373
firefox	44.0	173095	37887	Notepad++	6.8.8	7796	2015
fltk	1.3.2	7587	2833	Notepad++	6.8.7	7768	2009
FileZilla	3.27.0.1	97	7701	nspr	4.10.2.0	881	181
glew	1.5.1.0	563	306	nss	27.0.1.5156	5979	1745
Hasher	1.7.0	232	183	openssl	0.9.8	1376	415
hashdeep	4.3	3096	965	pcre3	3.9.0.0	52	48
info-zip/funzip	6.0	79	28	python	3.5.1	1538	28070
info-zip/zip	3.1	343	297	python	2.7.1	358	18200
info-zip/unzip	6.0	230	231	putty/putty	0.66 beta	1506	512
ibavcodec	11.10	719	99875	putty/plink	0.66 beta	1057	332
jsoncpp	0.5.0	1056	13	putty/pscp	0.66 beta	1157	344
lcms	8.0.920.14	668	182	putty/psftp	0.66 beta	1166	352
libcurl	10.2.0.232	1456	427	Qt5Core	2.0.1	17723	3987
libgd	1.3.0.27	883	497	SQLite	11.0.0.379	1252	307
libgmp	0.0.0.0	750	669	tinyXML	2.0.2	533	147
libjpeg	0.0.0.0	352	133	TestSSL	4	565	186
libpng	1.2.51	202	60	TrueCrypt	7.2	1193	2514
libpng	1.2.37	419	254	ultraVNC/vncviewer	1.2.13	4410	2045
libssh2	0.12	429	115	Winedt	9.1	87	8617
libtheora	0.0.0.0	460	226	WinMerge	2.14.0	405	6283
libtiff	3.6.1.1501	728	432	Wireshark	2.0.1	70502	39658
libxml2	27.3000.0.6	2815	1021	xampp	5.6.15	5594	111436

from their official websites (e.g., WireShark); the compiler of these binaries are detected by a packer tool called ExeinfoPE [3]. We evaluate our approach on a set of binaries, where a subset of them is detailed in Table 4.2.

4.4.2 Evaluation Metrics

Our ultimate goal is to discover as many relevant functions as possible with less concern about false positives, which means that recall has higher priority than precision. Hence, we choose to use the F_2 measure because it weights recall twice as heavily as it weights precision. The precision, recall, false positive rate (FPR), total accuracy (TA) and F_2-measure metrics are defined as follows [54]:

$$Precision = \frac{TP}{TP+FP},\ Recall = \frac{TP}{TP+FN},\ FalsePositiveRate(FPR) = \frac{FP}{FP+TN}$$

$$TotalAccuracy(TA) = \frac{TP+TN}{TP+TN+FP+FN},\ F_2 = 5.\frac{Precision.Recall}{4Precision+Recall} \tag{4.8}$$

where TP indicates number of relevant functions that are correctly retrieved; FN presents the number of relevant functions that are not detected; FP indicates the number of irrelevant functions that are incorrectly detected; and TN returns the number of irrelevant functions that are not detected.

In our experimental setup, we split the collected binaries into ten sets, reserving one as a testing set and using the remaining nine sets as the training set. We repeat this process 100 times and report the average output of the system in terms of aforementioned evaluation metrics. These metrics are calculated at function level (Section 4.3.1) and at project level (Section 4.3.2 and Section 4.4).

4.4.3 Accuracy of *FOSSIL*

In this subsection, we test *FOSSIL* in the context of several scenarios: (i) examining the effect of Bayesian network model; (ii) examining accuracy across different versions of FOSS packages; and (iii) comparing *FOSSIL* with existing state-of-the-art solutions.

A. Effect of Bayesian network model

We evaluate the accuracy of our system by examining it on a randomly collected binaries compiled with VS 2010, VS 2012, and GCC compilers from our repository. We test our system and report the precision, recall, and F_2 measure metrics. The results are summarized in Table 4.3, without and with the use of Bayesian network (BN) model.

Table 4.3: Effect of Bayesian network model

	Without a BN model			With a BN model		
Features	**Prec.**	**Rec.**	F_2	**Prec.**	**Rec.**	F_2
Opcodes	0.76	0.80	0.79	0.82	0.86	0.85
CFG-walks	0.72	0.76	0.75	0.84	0.83	0.83
Opcode distributions	0.70	0.72	0.71	0.81	0.86	0.85
Average/ All together	0.727	0.76	0.75	0.93	0.84	0.86

B. Accuracy across different versions of FOSS packages

We are further interested in evaluating *FOSSIL* with different versions of FOSS packages. For this purpose, we collect three different versions of all 160 projects in our repository, compile them with VS 2010 and test *FOSSIL*. The average output of the system in terms of precision, recall, and F_2 measure metrics is reported in Table 4.4. The highest obtained F_2 measure is 0.863 which is related to `openssl`, and the lowest one is 0.727 for `lcms`. The low F_2 measure for `lcms` can be attributed to the presence of many small functions that have been inlined.

Accuracy Interpretation. Our results demonstrate the following points:

1. *Pre-processing*: Some of the top-ranked opcode features are related to the compiler functions (e.g., stack frame setup operations). It is thus necessary to filter

Table 4.4: Accuracy results of different versions of FOSS packages

Project	Prec.	Rec.	F_2	Project	Prec.	Rec.	F_2
SQLite	0.78	0.81	0.803	libxml2	0.76	0.78	0.775
Webph	0.80	0.74	0.751	libjsoncpp	0.84	0.83	0.831
Xterm	0.79	0.81	0.805	Mersenne Twister	0.81	0.79	0.793
Hashdeep	0.81	0.85	0.841	libssh2	0.80	0.79	0.791
TinyXML	0.79	0.74	0.749	openssl	0.83	0.88	**0.863**
libpng	0.77	0.79	0.785	bzip2	0.79	0.80	0.797
ultraVNC	0.73	0.80	0.785	UCL	0.73	**0.9**	0.859
lcms	0.81	0.71	0.727	TrueCrypt	0.77	0.79	0.785
libavcodec	0.80	0.82	0.815	liblivemedia	0.80	0.81	0.807
info-zip	0.76	0.79	0.783	Libavutil	**0.84**	0.86	0.855
Firefox	0.77	0.81	0.802	Expat XML parser	0.80	0.8	0.8

out compiler functions to ensure better precision. Accordingly, in future work, we will leverage BINCOMP [108] with the current version of *FOSSIL* to distinguish compiler-related functions and FOSS-related functions. This will lead to considerable time savings and help shift the focus of the analysis to more relevant functions.

2. *Project Type*: We found that the accuracy of *FOSSIL* depends on the type of projects. For instance, in our experiments, *FOSSIL* achieves high accuracy when it discovers cryptography libraries since these libraries generally have more arithmetic and logical operations. Also, *FOSSIL* is able to identify unique CFG-walks that are related to certain cryptography operations. In contrast, we found that the accuracy of *FOSSIL* is slightly lower when it deals with parser libraries because they have functionalities in common with other libraries. For instance, `libucl` parser has common functionality with `JSON`; moreover, it can be integrated with a scripting language, such as `lua`. In Table 4.4, reasonably good precision is observed for `openSSL`, while the precision for `libxml2` is 0.76. To tackle the effects of

project type, we have to integrate more semantic features, e.g., type inference.

3. *Project Size*: We observed through experiments that the accuracy of our system is not affected by the size of the function or of the project. For example, a comparison of the precision achieved by FOSSIL for `Firefox` and `openSSL` (0.77 and 0.83, respectively) with that of `libpng` and `bzip2` (0.77 and 0.79, respectively) illustrates that our features can be extracted regardless of the size of functions, and that they can reveal the semantics of any piece of code regardless of its size.

4. *Features Extraction Level*. Typically, existing methods extract features from only one code level: instruction, function, or program level. A great advantage of FOSSIL is that it extracts features from all levels, making it possible to discover a function through different aspects. Also, the effect of code transformations such as the use of different compilers is reduced. In addition, we leverage concepts from biology to both extract the semantics of structural features and to improve efficiency when we deal with structural features.

5. *Parameter Selection*. For the Bayesian network model, *FOSSIL* uses three parameters Ψ_1, Ψ_2, and Ψ_3, with values of 0.45, 0.35, and 0.2, respectively. Applying different values to the Bayesian network model makes it possible to achieve various trade-offs between precision and recall, as shown in Figure 5. Tuning these parameters may result in different values for precision and recall.

4.4.4 Comparison

We compare our system to existing state-of-the-art systems: IDA FLIRT, RENDEZVOUS, [81], SARVAM [101], BINCLONE [63], TRACY [26,53], SIGMA [29], and LIBV [107]. The code of all aforementioned systems are available, with the exception of RENDEZVOUS. We re-implement RENDEZVOUS with paying special attention to the definition of its characteristics as well as its stated assumptions.

Table 4.5: Statistics about FLIRT signatures on the FOSS packages

Category	**No. of Signatures**	**Example**
Compression	300	E.g., Zlib, Bzip, UCL, infozip
Encryption	313	E.g., Botan, OpenSSL, TrueCrypt
Graphics	351	E.g., bgfx, openVDB, libpng
Web browser	307	E.g., crow, libOnion, firefox
Parsing	280	E.g., Expat, LibXml, TinyXml
Multimedia	171	E.g., LibVLC, SDL,
Database	178	E.g, MySQL++, SQLite, LMDB++, redis3m
JSON	204	E.g., json, jbson, libjson, jsonCPP
Networking	591	E.g., Restbed, Libcurl, Putty, WebSocket
Scripting	222	E.g., glew, lua
Math	70	E.g., libgmp
Editors	76	E.g., Notepad++
Hashing	152	E.g., Hashdeep, pHash, blockhash
Total	**3215**	

It is worthy to note that since FLIRT is a signature-based technology, for the sake of comparison, it is required to create a set of signatures for the projects being evaluated. To this end, we employ FLAIR technology [19], though the process is not fully automated and is considered a time-consuming task. Certain statistics regarding the FLIRT signatures generated by FLAIR are shown in Table 4.5. The number of functions in the FOSS package corpus for which FLIRT had signatures is 457, which is approximately 14% of the total created signatures. This low percentage can be explained by the main goal of

FLIRT technology, which is to identify the standard library functions such as C-standard libraries. In addition, the percentage of signature collision is 19%, which must be fixed by extending the signature formed; this further increases time consumption. Since each of the existing systems use different metrics to measure the accuracy, we unify the metric by using precision, recall, total accuracy (TA), and false positive rate (FPR). The obtained accuracy results on some projects as well as ROC curve of all projects are shown in Table 4.6 and Figure 4.3, respectively.

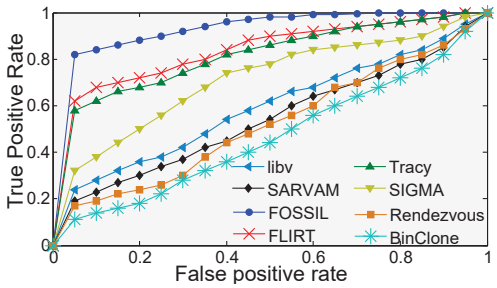


Figure 4.3: ROC curve

As can be seen, our system effectively identifies FOSS functions in the selected projects, and returns an average of 95% precision and 89% recall. Its accuracy is superior to that of the other systems, including FLIRT technology, which achieves the second highest rate of average precision 91% and 78% recall, as well as TRACY, which yields in some projects the highest rate of precision of 82%. On the other hand, BINCLONE achieves the lowest true positive rate since it employs exact matching, which causes a high rate of false positives. The reason of increase in precision rate in FOSSIL could be because of the combination types of the features (e.g., semantic and syntactic). In addition, ranking process helps to reduce the general and irrelevant opcodes in order to

increase the accuracy. Moreover, our system employs Bayesian network model that can control false positives rates by defining three threshold values.

An analysis of other systems reveals various limitations. TRACY is more accurate compared with the other systems, since data flow constraints are applied on `tracelets` (decomposing CFGs into subtraces of fixed length, excluding jump instructions). However, TRACY assumes that the candidate function should contain at least 100 basic blocks [53]; otherwise, it has a high rate of false positives. SIGMA integrates different graph representations, such as register flow graph, control flow graph, and call graph, to represent more semantics, whereas the approach is computationally expensive. The features used by RENDEZVOUS, such as `n-grams` and `k-CFGs`, are sensitive to code changes that lead to more false positive rates. Although grayscale images used by SARVAM are rich sources of information, they include many irrelevant features that increase the rate of false positives. LIBV generates execution dependence graphs (EDG) by applying data and control flow constraints; however, some issues such as having isomorphic EDGs for two different functions affect the accuracy.

Performance. We also compare the performance of each system by computing the overall execution time, which involves the feature extraction, and searching through the repository to find matches. The purpose of measuring performance is to evaluate the practicality of each system for large-scale datasets. For this purpose, no time limit is set to finish the FOSS function identification. However, we notice that some approaches such as BINCLONE, SIGMA, and LIBV are taking long time to detect functions since they are not

Table 4.6: Accuracy results of different existing approaches. (TA): total accuracy, (FPR): false positive rate, (Prec.): precision, and (Rec.): recall

		FireFox	Zlib	Jsoncpp	Libpng	OpenSSL	Python	Wireshark	Curl	TinyXML	Xaamp
TA	FLIRT	0.56	0.66	0.74	0.75	0.83	0.84	0.70	0.82	0.76	0.76
	TRACY	0.55	0.66	0.77	0.65	0.73	0.82	0.76	0.61	0.50	0.60
	SIGMA	0.65	0.69	0.74	0.70	0.71	0.82	0.76	0.62	0.48	0.58
	BINCLONE	0.49	0.63	0.63	0.64	0.66	0.69	0.76	0.58	0.42	0.60
	RENDEZVOUS	0.53	0.65	0.64	0.65	0.63	0.68	0.65	0.53	0.39	0.52
	SARVAM	0.69	0.74	0.76	0.69	0.73	0.84	0.75	0.66	0.51	0.6
	LIBV	0.49	0.57	0.66	0.62	0.69	0.72	0.77	0.62	0.42	0.60
	FOSSIL	0.98	0.80	0.93	0.84	0.90	0.85	0.90	0.87	0.81	0.79
FPR	FLIRT	0.09	0.26	0.13	0.18	0.49	0.32	0.26	0.34	0.26	0.39
	TRACY	0.35	0.51	0.36	0.22	0.48	0.51	0.35	0.52	0.25	0.46
	SIGMA	0.36	0.53	0.36	0.24	0.49	0.51	0.37	0.52	0.25	0.46
	BINCLONE	0.36	0.67	0.51	0.27	0.54	0.45	0.41	0.55	0.45	0.46
	RENDEZVOUS	0.37	0.61	0.43	0.27	0.52	0.44	0.53	0.55	0.45	0.46
	SARVAM	0.39	0.53	0.35	0.20	0.49	0.51	0.38	0.52	0.21	0.48
	LIBV	0.49	0.54	0.46	0.31	0.44	0.42	0.45	0.48	0.50	0.56
	FOSSIL	0.15	0.28	0.25	0.19	0.19	0.26	0.39	0.38	0.18	0.39
Prec.	FLIRT	0.93	0.88	0.95	0.93	0.90	0.94	0.89	0.93	0.92	0.88
	TRACY	0.77	0.78	0.90	0.86	0.84	0.89	0.90	0.72	0.77	0.74
	SIGMA	0.84	0.79	0.89	0.88	0.83	0.89	0.88	0.73	0.75	0.72
	BINCLONE	0.73	0.70	0.76	0.84	0.77	0.81	0.87	0.69	0.55	0.74
	RENDEZVOUS	0.75	0.72	0.79	0.85	0.75	0.80	0.75	0.64	0.5	0.66
	SARVAM	0.85	0.82	0.89	0.89	0.84	0.90	0.87	0.76	0.79	0.73
	LIBV	0.66	0.68	0.80	0.81	0.81	0.84	0.87	0.73	0.52	0.70
	FOSSIL	0.99	0.93	0.98	0.96	0.98	0.95	0.96	0.94	0.95	0.90
Rec.	FLIRT	0.58	0.72	0.75	0.76	0.90	0.88	0.73	0.87	0.8	0.82
	TRACY	0.65	0.78	0.83	0.64	0.83	0.90	0.81	0.74	0.53	0.71
	SIGMA	0.74	0.81	0.80	0.69	0.82	0.90	0.82	0.75	0.51	0.69
	BINCLONE	0.60	0.81	0.77	0.66	0.80	0.78	0.82	0.73	0.50	0.70
	RENDEZVOUS	0.63	0.80	0.74	0.67	0.77	0.76	0.77	0.68	0.45	0.62
	SARVAM	0.78	0.84	0.81	0.67	0.83	0.91	0.81	0.78	0.50	0.71
	LIBV	0.65	0.72	0.77	0.65	0.77	0.79	0.85	0.72	0.52	0.75
	FOSSIL	0.99	0.84	0.94	0.85	0.91	0.88	0.93	0.90	0.82	0.85

scalable enough to obtain the search result within a specific given time frame.

In particular, the execution time for FOSS function identification in *FOSSIL* was measured by adding the time required for each step (normalization, opcode ranking, and feature extraction in each component) to the time spent to discover the FOSS functions. Feature extraction in the first component takes 5 sec for the small packages in our dataset (e.g., 100 functions) and 15 sec for the large package (e.g., 50,000 functions). The

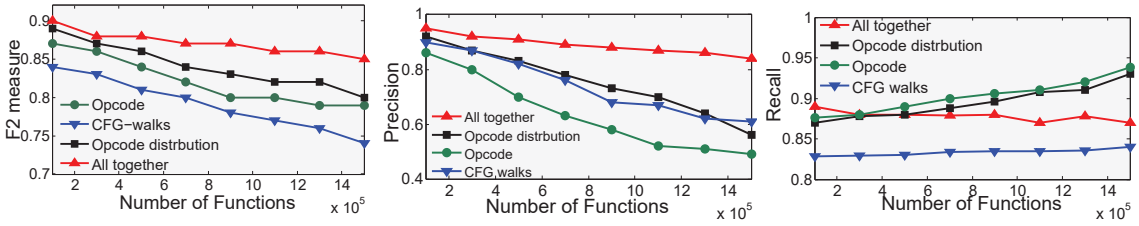
proposed hash subgraph kernel is fast, taking an average of 5 sec for all packages in a similar environment. The time required to extract features in the third component is negligible (less than 1 ms). Our system spends the majority of time on searching the repository; further optimizing the search using advanced indexing techniques is a future direction. Each search iteration takes a minimum of 7 sec and a maximum of 50 sec.

The overall time for FOSSIL ranges from 17 to 80 sec, while the averages for RENDEZVOUS, TRACY, SARVAM, SIGMA, and LIBV are 72.5, 115, 55, 155, and 111.5 sec, respectively. We observe that the performance of SARVAM is closer to that of *FOSSIL*, since the extraction of image features is relatively efficient. The performance of RENDEZVOUS is also close since it uses a Bloom filter, which speeds up the retrieval process.

4.4.5 Scalability Study

Since one of our ultimate goals is to build a searchable index for large-scale FOSS projects based on the proposed approach in this chapter, in addition to time efficiency, we evaluate the scalability of FOSSIL when it is used to index and retrieve matched functions on a large number of projects. This made it possible to investigate the trade-off between accuracy and efficiency. For this purpose, we add more projects, dlls, operating system applications, and other programs to our repository. In total, there are 500 applications and approximately 1.5 million functions. We measure the total time required to index the project and to match the target files. In addition, we examine the accuracy of each component separately and all together. Figure 4.4 shows that our system is scalable when

the number of functions reaches to 1.5 million.



(a) (b) (c)

Figure 4.4: Performance of *FOSSIL* against a large set of functions

The F_2 measure falls down slightly, from 0.9 to 0.86, which provides some insight into the scalability of system when it deals with a large number of FOSS functions. Based on these results, we believe our system will be efficient and practical for most real-world applications.

4.4.6 Confidence Estimation of Bayesian Network

Using a Bayesian network model provides a confidence estimator based on probability scores, where higher probability scores correspond to higher confidence. Future research will hopefully produce an actual probability score. Applying different factor values to the Bayesian network model makes it possible to achieve various trade-offs between precision and recall. Figure 4.5 shows the results of confidence estimation for three factors, varying the trade-off between precision and recall. A precision measure of 50% is achieved with a recall measure of just under 80%; conversely, 50% recall gives over 80% precision.

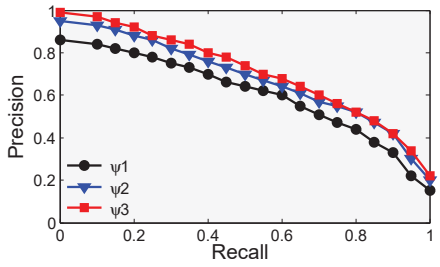


Figure 4.5: Confidence estimation: precision vs. recall

4.4.7 Impact of Evading Techniques

We consider the projects from our dataset in order to test FOSSIL against binary and source obfuscation, as shown in Table 4.7. The obfuscation process is done in two stages. First, C^{++} refactoring tools [1, 16] are used for source code level obfuscation. These rely on the following techniques: *moving a method from a superclass to its subclasses*, and *extracting a few statements and placing them in a new method*. We refer the reader to [67] for in-depth explanations of these techniques. We also apply Nynaeve [21] tool, which comprises *Frame Pointer Omission* and *Function inlining* methods. These methods are described in 2.3.

Second, to investigate binary-level obfuscation, we compile the 160 projects with GCC and VS compilers, and the resulting binaries are obfuscated using DaLin [87] generator and Obfuscator-LLVM [78]. These obfuscators replace instructions with other semantically equivalent instructions (*instruction substitution*). Obfuscator-LLVM also applies *control flow flattening*, and *bogus control flow* techniques, whereas DaLin performs *instruction reordering*, *dead code insertion*, and *register renaming* as well. The obfuscated binaries are passed as target binaries to our system, and we then measure the

Table 4.7: Evading technique tools, methods, and their effects on FOSSIL components

Tool	Method	Input	Output	A^*	Component		
					Opcode	CFG-Walk	Opcode Dist.
LLVM [77]	CFG flattening	Bin	Bin	74%	○	●	○
	Instruction substitution			84%	●	○	○
	CFG bogus			81%	○	●	○
DaLin [87]	Instruction reordering	Asm	Asm	86%	○	○	○
	Dead code insertion			86%	○	○	○
	Register renaming			86%	○	○	○
	Instruction substitution			84%	●	○	○
Trigress [25]	Virtulazation	Src	Src	82%	●	●	●
	Jitting			83%	●	○	○
	Dynamic			83%	●	○	○
PElock [23]	Hide procedure call	Asm	Asm	86%	○	○	○
	Insert fake instruction			86%	○	○	○
	Prefix junk opcode			86%	○	○	○
	Insert junk handlers			86%	○	○	○
Nynaeve [21]	Frame pointer omission	Src	Bin	81%	●	●	○
	Function inlining			80%	○	●	○
OREANS [22]	Binary encryption	Src	Bin	NA	NA	NA	NA
Gas Obfuscator [17]	Junk byte	Asm	Asm	86%	○	○	○
Designed Script	Loop unrolling	Src	Src	77%	●	○	●

Note: (A^*) indicates accuracy after applying obfuscation method while the accuracy before applying obfuscation method is 86%. (○) indicates there is no effect, while (●) means the corresponding component get affected. (*NA*) means not applicable. We use the following abbreviations: (Bin) Binary, (Asm) Assembly, (Src) Source, and (Dist.) Distribution.

accuracy of function identification.

Our system obtains an average F_2 measure of 83.1% in identifying similar FOSS functions, which represents only a slight drop in comparison to the 86% observed without obfuscation.

As can be seen in Table 4.7, the obfuscation tools work at three levels: source, binary, and assembly. It can be observed that the obfuscation methods of *CFG flattening*, *function inlining*, and *loop unrolling* decrease the accuracy of FOSSIL by approximately $6 - 12\%$. However, their effect on accuracy is not significant since FOSSIL employs a Bayesian network to synthesize the knowledge obtained from the three components by defining a confidence estimator function. Table 4.7 also shows that FOSSIL cannot deal with encrypted binaries. The current version of FOSSIL consists of components relying

on static analysis. A possibility for future work is to extend FOSSIL by including dynamic components that can deal with encrypted binaries.

There are three main reasons for the slight drop in accuracy. The first component, HMM, deals with opcode frequencies at the function level, so in the case of *instruction reordering*, all the reordered instructions, regardless of order, will be captured. In addition, since the operands are not considered in this component, *register renaming* does not affect the accuracy. However, this component is affected slightly by *instruction replacement* because this technique affects frequencies. However, as previously mentioned, the chi-squared test is used to evaluate the frequencies, and it involves a confidence interval that varies according to user requirements. The second component, CFG-walk, tolerates instruction-level obfuscation to a greater extent since it deals with the semantics of a function as well as the instruction groups. To avoid *bogus control flow* and *function inlining* techniques, we use the most important opcodes to color CFG-walks. We also label a node with its neighbors in a novel way in order to avoid any obfuscation that can affect the CFG. However, this component is affected by *CFG flattening*. The third component, z-score, measures the area of the opcode distribution, so both *instruction replacement* and *dead code insertion* may slightly affect it. In general, using opcode ranking, normalization, and coloring techniques reduce the effects of most aforementioned obfuscation methods. However, the Bayesian network model synthesizes the knowledge obtained from the three components; therefore, if the knowledge from one is not sufficient, the Bayesian network model will automatically assign more weight to the other components.

The Impact of Compilers. To create an experimental dataset, we consider 160 projects compiled with Visual Studio (VS), GNU Compiler Collection (GCC), Intel C++ Compiler (ICC), and Clang compilers with Od optimization setting. To measure the effect of different compilation options such as compiler optimization flags, we additionally compile them with level-1, level-2, and level-3 optimizations, namely the $O0$, $O2$, and Ox flags. We extract features for each compilation setting, and then test our system. The results illustrated in Table 4.8 show that the features extracted by our system are greatly effective for most optimization speed levels.

Table 4.8: FOSS function identification with different compilers and compilation settings

Compiler	Optimization Speed	Precision	Recall
VS	$O0$, $O2$, Ox	0.95, 0.95, 0.95	0.94, 0.92, 0.92
GCC	$O0$, $O2$, Ox	0.92, 0.92, 0.92	0.93, 0.90, 0.89
ICC	$O0$, $O2$, Ox	0.78, 0.74, 0.69	0.81, 0.80, 0.78
Clang	$O0$, $O2$, Ox	0.65, 0.59, 0.60	0.64, 0.60, 0.58

The normalization process used in our system can reduce the effect of GCC and VS compilers. Moreover, the top-ranked opcodes are more related to the semantics of the function, in addition to the colored CFG-walks which help to avoid compiler effects. However, the accuracy drops significantly when the source compilers are Clang or ICC, since these compilers produce more optimized code compared to VS and GCC compilers. Such limitations can be handled by first identifying the compiler using existing tools such as Exeinfo [3] and then applying the suitable features accordingly.

4.5 Summary

To conclude, we have conducted the first investigation in identifying FOSS functions. we proposed a novel resilient and efficient system that incorporates three components. The first component extracts the syntactical features of functions by considering opcode frequencies and applying a hidden Markov model statistical test. The second component applies a neighborhood hash graph kernel to random walks derived from control flow graphs, with the goal of extracting the semantics of the functions. The third component applies z-score to the normalized instructions to extract the behavior of instructions in a function. The components are integrated using a Bayesian network model which synthesizes the results to determine the FOSS function. Our evaluation demonstrates that our proposed system yields highly accurate results.

Chapter 5

Identifying the Authors of Program Binaries

5.1 Overview

In this chapter, we present *BinAuthor,* a system that extracts authors' coding habits from binary code. To capture coding habits, *BinAuthor* leverages a set of features that are based on collections of functionality-independent choices made by authors during coding. Our evaluation shows that *BinAuthor* outperforms existing methods in several aspects. First, it successfully attributes a larger number of authors with a significantly higher accuracy when compared to existing research contributions. Second, *BinAuthor* is more robust than previous methods in the sense that there is no significant drop in accuracy when the code is subjected to refactoring techniques, code transformation, and different compilers.

5.2 Preliminaries

In this section, we first provide an overview of the main idea and then introduce the threat model.

5.2.1 Authorship Attribution

The feasibility of authorship attribution generally relies on the fact that software developers usually possess certain coding habits, which may be influenced by their education and training, experiences, development environments, etc. For example, a programmer coming from a procedural programming background may be more reluctant to take advantage of object-oriented modularization than person who starts to learn programming with an object-oriented language. Although one coding habit is usually not enough to uniquely identify an author, a collection of such habits may be sufficient. However, there are two key challenges in recognizing such coding habits from a binary. First, *how to ensure that the extracted features represent coding habits instead of something dominated by the program's functionality?* Second, *which coding habits are preserved in the binaries after the compilation process?*

To address the first issue, our main idea is twofold: First, we rely on coding habits that can be represented as functionality-independent choices; for instance, the same functionality could be implemented by either in a procedural programming or object-oriented styles, with either more global variables or more local variables, with more `while` loops or more `for` loops, etc. Second, in addition to relying on functionality-independent

choices, we also ensure that these choices are habitual enough for each author, by only considering those that frequently appear across various program binaries performing different tasks written by that author. As a result, only those functionality-independent choices that are also persistent for each author regardless of varying functionality will be considered as candidate features. To address the second issue, we investigate a large collection of source code together with their mapping to assembly instructions to determine which candidate features are preserved in the binaries. In Section 5.3, we show how *BinAuthor* captures such coding choices using a rich list of features.

5.2.2 Threat Model

BinAuthor is a system specifically designed to assist reverse engineers in discovering information from a binary that may be related to its author(s). As such, it is not intended for general purpose reverse engineering tasks, such as unpacking or deobfuscating malware samples. We investigate refactoring and code transformations later in this Chapter in order to demonstrate possible evading countermeasures that may be used by future malware authors in order to circumvent detection. More specifically, the threat model of this chapter is further clarified in what follows.

Since the research on binary authorship attribution is still in its infancy, *BinAuthor* is certainly not meant as a bullet-proof solution. As such, strengthening it against possible countermeasures will be an ongoing and continuous battle. Nonetheless, we dedicated special care to evading techniques while designing and implementing *BinAuthor*. We have

taken into consideration some potential evading techniques. First, we assume that the adversaries might apply refactoring techniques to evade detection. Second, the adversaries might apply code transformations to source and binary files. Third, since a program binary can be significantly changed by simply choosing a different compiler or by altering the compilation settings. Finally, the adversaries may intentionally avoid or fake some of their coding habits (however, it is known to be hard for a programmer to avoid all coding habits or use a different style for each program, and in fact, faking a style may even help detection in some cases [91]).

We show how *BinAuthor* survives the first three aforementioned threats in Section 5.4. As for the last threat, the features of *BinAuthor* have been carefully designed to capture coding habits at multiple abstraction levels, which makes it harder for adversaries to evade detection even if they are aware of the habits being looked for [91]. In addition, an operational solution is to customize and enrich the list of features based on the actual use case and learning data, which will both improve accuracy and make it more difficult for adversaries to hide all their habits.

5.3 BinAuthor

Our goal is to automatically identify the author(s) of binary code. We approach this problem using different distance metrics; that is, we generate a list of functionality-independent choices from training data of sample binaries with known authors. Hence, we propose a system encompassing different components, each of which is meant to achieve

a particular purpose. The first component, (*Filtration*), isolates user functions from compiler functions and library functions. Hence, the outcome of this component is considered as a habit (e.g., the preference in using specific compiler or free software packages). For instance, using `GCC` compiler rather than `visual studio`, or using `SQLite` rather `MongoDB`, etc. The second component, (*Feature Categorization*), analyzes binary code to extract possible features that represent stylistic choices. The third component, (*Author Habits Profiling*), constructs a repository of habits of known authors. The last component (*Authorship Attribution*) performs matching to *BinAuthor*'s repository for author classification attribution. Figure 5.1 illustrates the architecture of *BinAuthor*. The aforementioned components are explained in depth in the remainder of this section.

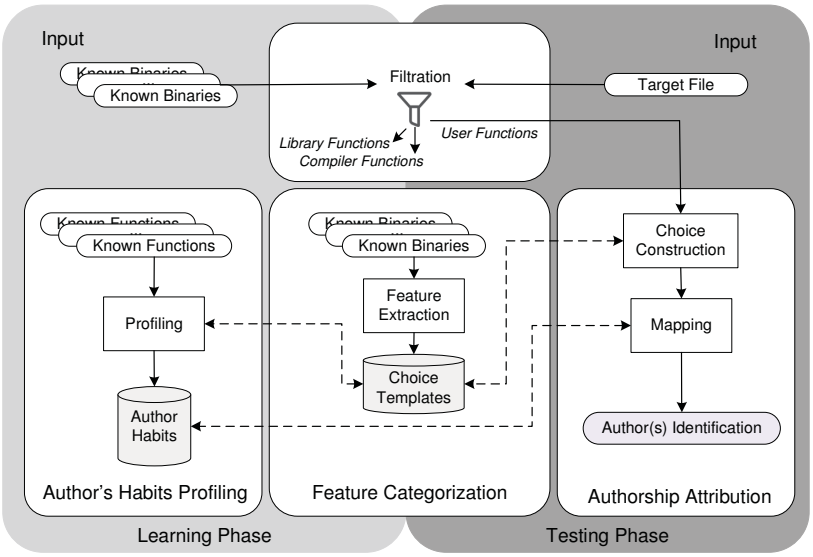


Figure 5.1: *BinAuthor* architecture

5.3.1 Filtration Process

An important initial step in most reverse engineering tasks is to distinguish between user functions and library/compiler functions. This step saves considerable time and helps shift the focus to more relevant functions. The filtration process consists of three steps. First, FLIRT [6] (Fast Library Identification and Recognition Technology) technology is used to label library functions. Second, a set of signatures is created for specific FOSS libraries such as `Sqlite3`, `libpng`, `zlib`, and `openssl` using Fast Library Acquisition for Identification and Recognition (FLAIR) [6]; this set is added to the existing signatures of the IDA FLIRT list. The last step performs compiler functions filtration, the details of which are explained below. Also, we employ our proposed work in Chapter 3 to identify reused functions. Further, we use FOSSIL, our proposed work in Chapter 4, in order to identify FOSS-related functions.

The idea is based on the hypothesis that compiler/helper functions can be identified through a collection of static signatures that are created in the training phase (e.g. opcode frequencies). We analyze a number of programs with different functionality, ranging from a simple “Hello World!" program to programs fulfilling complex tasks. Through the intersection of these functions combined with manual analysis, we collect *120* functions as compiler/helper functions. The opcode frequencies are extracted from these functions, after which the mean and variance of each opcode among all opcodes are calculated. In other words, each disassembled program *P*, after passing FLIRT, consists of *n* functions $\{f_1, \cdots, f_n\}$. Each function f_k is represented as m pairs of opcodes o_i, where m is the

number of distinct opcodes in function f_k. Each opcode $o_i \in O$ has a pair of values (μ_i, ν_i), which represents the mean and variance values of that specific opcode. Each opcode in the target function is measured against the same opcode of all compiler functions in the training set. If the measured distance $D_{i,j}$ is less than a predefined threshold value $\alpha = 0.005$, the opcode is considered as a match. A function is labeled as *compiler-related* if the matched opcodes ratio is greater than a predefined threshold value learned from experiments to be $\gamma = 0.75$; otherwise, the target function is labeled as *user-related*. Similarity measurements are performed based on distance calculations as per the following definition [123]:

$$D_{i,j} = \frac{(\mu_j - \overline{\mu_j})^2}{\left(\nu_i^2 + \overline{\nu_i^2}\right)}$$

where ($\overline{\mu}_j$, $\overline{\nu}_j$) represents the opcode mean and variance, respectively, of the target function. This similarity detects functions, which are closer to each other in terms of types of opcodes. For instance, logical opcodes are not available in *compiler-related* functions. Finally, a score is given to every distance that is below a predefined threshold α. A function is tagged as a *compiler-related* function if the ratio of summation of scores to the number of unique opcodes is beyond a given threshold γ; otherwise, the target function is deemed a *user-related* function.

5.3.2 Feature Categorization

Determining a set of characteristics that remain constant for a significant portion of programs, written by an author, is analogous to finding human characteristics that can later be used for the identification of a specific person. To this end, our aim is to automate the finding of such program characteristics, and with reasonable computational cost. To capture coding habits at different abstraction levels, we consider a spectrum of such habits, as illustrated in Figure 5.2. As shown in Figure 5.2, an author's habits can be reflected in the preference of choosing certain keywords or compilers, the reliance on the main function, or the use of an object oriented programming paradigm. In addition, the manner in which the code is organized by the author may also reflect author habits. All possible choices are stored as a template in this step. Moreover, we introduce a novel taxonomy of coding habits in Figure 5.2. We provide details of each category of functionality-independent choices in the following subsections.

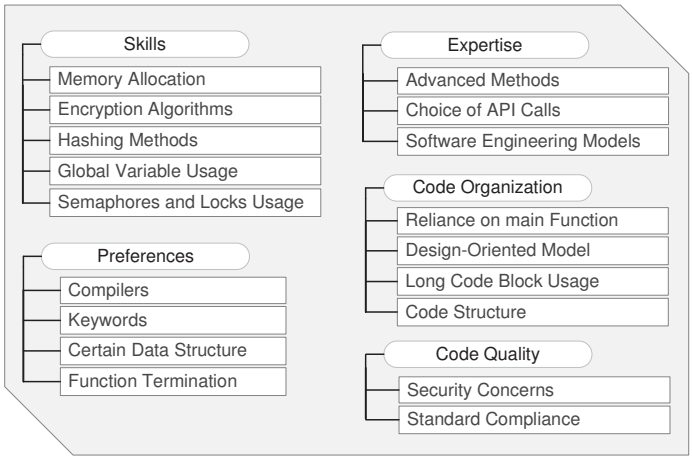


Figure 5.2: Coding habit taxonomy

A. General Choices

General choices are designed to capture the author's general programming preferences; for example, preferences in organizing the code, terminating a function, use of particular keywords, or use of specific resources.

1) Code Organization: We capture how code is organized by measuring the reliance on the `main` function since it is considered a starting part for managing user functions. We define a set of ratios, as shown in Table 5.1, that measures the actions used in the `main` function. We thus capture the percentage of usage of keywords, local variables, API calls, and calling user functions, as well as the ratio between the number of basic blocks of the `main` function to the number of basic blocks of other user functions. These percentages are computed in relation to the length of the `main` function, where the length signifies the number of instructions in the function. The results are represented as a vector of ratio values, which is used by the detection component.

Table 5.1: Features extracted from the `main` function: length(l): Number of instructions in the `main` function

Ratio Equation	Description
# of `push` / l	Ratio of local variables to length
# of `push` / # of `lea`	Ratio of local variables to memory address locations
# of `lea` / l	Ratio of memory address locations to length
# of calls / l	Ratio of function calls to length
# of indirect calls / l	Ratio of API calls to length
# of `BBs` / total # of all `BBs`	Ratio of the number of basic blocks of the `main` function to that of other user functions
# of calls / # of user functions	Ratio of function calls to the number of user functions

2) Function Termination: *BinAuthor* captures how an author terminates a function. This could help identify an author since programmers may be used to specific ways

of terminating a function. *BinAuthor* does not only consider the last statement of a function as the terminating instruction; rather, it considers the last basic block of the function with its predecessor as the terminating part. This is a realistic consideration since various actions may be required before a function terminates. To this end, *BinAuthor* not only considers the usual terminating instructions, such as `return` and `exit`, but also captures other related actions that are taken prior to termination. For instance, a function may be terminated with a display of messages, calling another function, releasing some resources, communication over networks, etc. Table 5.2 shows examples of what is captured in relation to the termination of a function. Each feature is set to *1* if it is used to terminate a function; otherwise, it is set to *0*. The output of this component is a binary vector that is used by the detection component.

Table 5.2: Examples of actions in terminating a function

Features	
Printing results to memory	Printing results to file
Using system ("pause")	User action such as `cin`
Calling user functions	Calling API functions
Closing files	Closing resources
Freeing memory	Flushing buffer
Using network communication	Printing clock time
Releasing semaphores or locks	Printing errors

3) Keyword and resource preferences: *BinAuthor* captures the author's preference of using different keywords or resources. We only consider groups of such preferences with equivalent or similar functionality in order to avoid functionality-dependent features. For instance, keyword type preferences for inputs (e.g., using `cin`, `scanf`), preferences of using particular resources or a specific compiler (we identify the compiler

through tracking strings called *compiler tags* [108]), and the manner in which certain keywords are used can serve as further indications of an author's habits.

General Choice Computation: We compute a set of vectors, v_{gi}, (where g represents general and i represents the sub-category number). To consider the reliance on the `main` function, a vector v_{g1}, representing related features, is constructed according to the equations shown in Table 5.1. These equations indicate the author's reliance on the `main` function as well as the actions performed by the author. Function termination is represented as a binary vector, (v_{g2}), which is determined by the absence or existence of a set of features for function termination. Keyword and resource preferences are identified through binary string matching, which tracks the annotations to `call` and `mov` instructions. For instance, excessive use of `fflush` will cause the string `"_imp_fflush"` to appear frequently in the resulting binary. We extract a collection of strings from all user functions of a particular author, then intersect these strings in order to derive a persistent vector (v_{g3}) for that author. Consequently, for each author, a set of vectors representing the author's signature is stored in our repository. Given a target binary, *BinAuthor* constructs the vectors from the target and measures the distance/similarity between these vectors and those in our repository. The v_{g1} vector is compared using Euclidean distance, whereas v_{g2} vector is compared using the Jaccard distance. For v_{g3}, the similarity is computed through string matching. Finally, the three derived similarity values are averaged in order to obtain λ_g, which is later used in Section 5.3.3 for the purpose of author classification.

B. Variable Choices

Developers often have their own habits for defining local and global variables, which may originate from the author's experiences or skills. The variable chain has been shown to greatly improve author attribution of source code [59]. It has been defined as the variable usage among different functions. Inspired by this work, we introduce a register chain to capture authors' habits in using variables. We define the register chain concept as the states of using a particular register through all basic blocks in a user function. To avoid compilation setting effects, we normalize the registers to general names such as Reg_1, Reg_2, etc. and keep their occurrence order. Useful characteristics of such chains include the longest chain, the shortest chain, the number of existing chains, the liveness of registers among basic blocks, etc.

Example: In what follows, we illustrate how a register chain is extracted. Part of the Control Flow Graph (CFG) of the `RC4` function in `Citadel` is shown in Figure 5.3(a). Figure 5.3(b) shows the construction of the register liveness [100] for the indicated registers.

As illustrated in Figure 5.3(b), the used registers `ecx`, `ebp`, `esi`, `ebx`, `edx`, and `al` are normalized to $Reg_1, \cdots, Reg_6$. The first, second, and third basic blocks manipulate $\langle Reg_1, Reg_2 \rangle$, $\langle Reg_3, Reg_4 \rangle$, and $\langle Reg2, Reg5, Reg6 \rangle$ registers, respectively. ***BinAuthor*** captures the register liveness by storing the set of basic blocks where the register is alive. For instance, the Reg_2 register appears in the first and third basic blocks and does not appear in the second basic block, so we represent the liveness of the Reg_2 register as

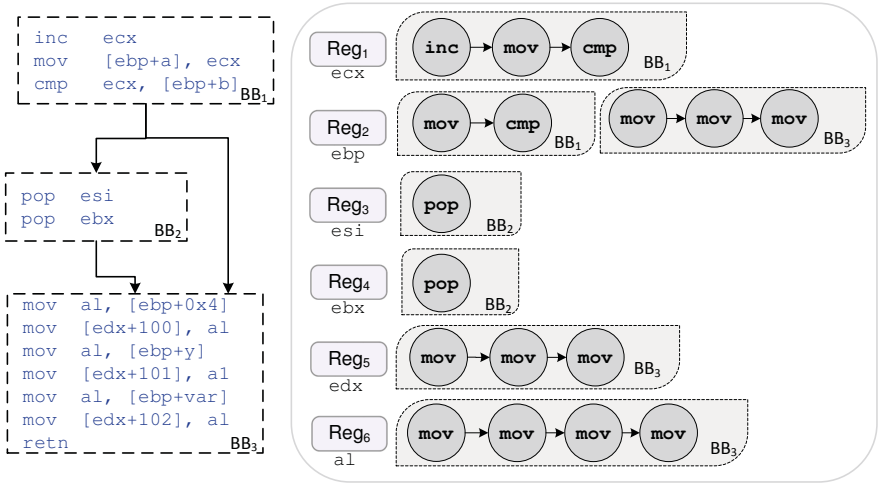


Figure 5.3: (a) Part of the CFG of RC4 (b) Register chain

$\{BB_1, BB_3\}$. A summary of the registers liveness is given in Table 5.3.

Table 5.3: Register liveness (✓indicates that the register is alive in a BB)

Register	BB_1	BB_2	BB_3
Reg_1	✓	-	-
Reg_2	✓	-	✓
Reg_3	-	✓	-
Reg_4	-	✓	-
Reg_5	-	-	✓
Reg_6	-	-	✓

Variable Choice Computation: Since each function may have a large number of register chains, *BinAuthor* employs locality-sensitive hashing (LSH) for feature reduction. The hash is calculated over all sets of chains, and only those with similar hash values are clustered (hashed) to the same bucket. In the case of register chain similarity, similar register chains will be hashed to the same bucket. Once register chains have been hashed to a corresponding bucket, any bucket containing more than one similar hash value is identified and a list of candidate register chains is extracted. Finally, similarity analysis

is performed to rank the candidate pairs obtained from the previous steps. The similarity score obtained from this choice is λ_v.

C. Quality-Related Choices

We investigate code quality in terms of standard compliance with C/C++ coding standards and security concerns. In the literature, code quality can be measured with different indicators, such as testability, flexibility, adaptability [109], etc. *BinAuthor* defines rules for capturing code that exhibits either relatively high or low quality. For any code that cannot be matched using such rules, the code is labeled as regular quality, which indicates that the code quality feature is not applicable.

Rules: Examples of low-quality coding styles include reopening already opened files, leaving files open when they are no longer in use, attempting to modify constants (i.e., through pointers), using float variables as loop counters, and declaring variables inside of a switch statement, which can result in unexpected/undefined behavior due to jumped-over instructions. Examples of high-quality coding styles include handling errors generated by library calls (i.e., examining the returned value by `fclose()`), avoiding reliance on side-effects (i.e., $++$ operator) within particular calls such as `sizeof` or `_Alignof`, averting the use of particular calls on some environments or using them with protective measures (i.e., the use of `system()` in Linux may lead to shell command injection or privilege escalation; hence, using `execve()` instead is indicative of high-quality coding), and the use of locks and semaphores around critical sections.

Quality-related Choice Computation: We build a set of idiom templates to describe high or low quality habits. Idioms are sequences of instructions with wild-card possibility [82]. We employ the idioms templates in [82] according to our qualitative-related choice. In addition, such templates carry a meaningful connection to the quality-related choices. Our experiments demonstrate that such idiom templates may effectively capture quality-related habits. *BinAuthor* uses the Levenshtein distance [131] for this computation due to its efficiency. The similarity is represented by λ_q, which is used in Section 5.3.3 for author classification purpose.

$$\lambda_q = 1 - \frac{L(C_i, C_j)}{max(|C_i|, |C_j|)}$$

where $L(C_i, C_j)$ is the Levenshtein distance between the qualitative-related choices C_i (sequence of instructions) and C_j, $max(|C_i|, |C_j|)$ returns the maximum length between two choices C_i and C_j in terms of characters.

D. Embedded Choices

We define embedded choices by actions that are related to coding habits present in the source code that are not easily captured at the binary level by traditional features such as strings or graphs. For instance, initializing member variables in constructors and dynamically deleting allocated resources in destructors are examples of embedded choices. As it is not feasible to list all possible features, *BinAuthor* relies on the fact that opcodes reveal

actions, expertise, habits, knowledge, and other author characteristics, and analyzes the distribution of opcode frequencies. Our experiments show that such a distribution can effectively capture the manner by which the author manages the code. As every single action in source code can affect the frequency of opcodes, *BinAuthor* targets embedded choices by capturing the distribution of opcode frequencies.

Example: In order to pass parameters to a function, a developer may choose to pass primitive types by value or to pass objects by reference, and may have preferences in using one particular algorithm over another. Such examples are not straightforward to be captured through tracking strings or CFGs. We observe through our experiments that embedded choices may share similar opcode distributions.

Embedded Choice Computation: For measuring the distance between distributions of opcode frequencies, the Mahalanobis distance [90] is used to measure the similarity of opcode distributions among different user functions. The Mahalanobis distance is chosen because it can capture the correlation between opcode frequency distributions, and this correlation represents the embedded choices. The similarity returned is represented by λ_e.

E. Structural Choices

Programmers usually develop their own habits in terms of structural design of an application. They may prefer to use a fully object-oriented design or they may be more accustomed to procedural programming. Such structural choices can serve as features for

author identification. To avoid functionality, we consider the common subgraphs and the longest path for each user function, and then intersect them among different user functions. These subgraphs are defined as k-graphs, where k is the number of nodes. These common k-graphs form author signatures since these graphs always appear regardless of the program functionality. In addition, we consider the longest path since it reflects how an author tends to use deep or nested loops.

Example: An author may organize different classes in an ad hoc manner, or organize them in a hierarchical way by designing a driver class to contain several manager classes, where each manager is responsible for different processes (a collection of threads running in parallel). Both ad hoc and hierarchical organizations will create a common structure in the author's programs.

Structural Choice Computation: *BinAuthor* uses subgraphs of size *k* in order to capture structural choices (*k* = 4, 5, and 6 through our experiments). Given a *k*-graph, the graph is transformed into strings using Bliss open-source toolkit [79]. Then, a similarity measurement is performed over these strings using the normalized compression distance (NCD) [48], which enhances search performance. We have chosen NCD since it allows us to concatenate all the common subgraphs that appear in author's programs. Additionally, it allows for inexact matching between the target subgraphs and the training subgraphs. *BinAuthor* forms a signature based on these strings. The similarity obtained from this choice is represented by λ_s.

5.3.3 Significance of BinAuthor Choices

As previously described, *BinAuthor* extracts different types of choices to characterize different aspects of author coding habits. Such choices do not equally contribute to the attribution process since the significance of these indicators are not identical. Consequently, a weight is assigned to each choice by applying logistic regression to each choice individually in order to predict class probabilities (e.g., the probability of identifying an author). The probability outcomes of logistic regression prediction is illustrated in Table 5.4. We calculate the weights as follows.

$$w_i = p_i / \sum_{i=1}^{5} rnd(p_i/p_s)$$

where p_s is the smallest probability value (e.g. 0.32 in Table 5.4), p_i is the probability outcome from logistic regression of each choice, and the rnd function rounds the normalized values (p_i/p_s), leading us to the weights shown in Table 5.4.

Table 5.4: Logistic regression weights for choices

Choice	**Probability**	**Weight**
General	0.83	0.33
Qualitative	0.63	0.22
Structural	0.52	0.22
Embedded	0.39	0.12
Variable	0.32	0.11

5.4 Evaluation

5.4.1 Implementation Setup

The described stylistic choices are implemented using separate Python scripts for modularity purposes, which altogether form our analytical system. A subset of the python scripts in the *BinAuthor* system is used in tandem with IDA Pro disassembler. The final set of the framework scripts performs the bulk of the choice analysis functions that compute and display critical information about an author's coding style. With the analysis framework completed, a graph database is utilized to perform complex graph operations such as k-graph extraction. The graph database chosen for this task is Neo4j [15]. Gephi [12] is employed for all graph analysis functions, which are not provided by Neo4j. MongoDB database is used to store our features for efficiency and scalability purposes.

5.4.2 Dataset

The used dataset is consisted of several applications from different sources, as described below: (i) GitHub [7], where a considerable amount of real open-source projects are available; (ii) Google Code Jam [5], an international programming competition, where solutions to difficult algorithmic puzzles are available; (iii) Planet Source Code [14], a web-based service that offers a large amount of source code written in different programming languages; (iv) Graduate Student Projects at our institution. Statistics about the dataset are provided in Table 5.5. In total, we test *152* authors from different sets in which

each author has two to ten software applications.

Table 5.5: Statistics about the dataset used in the evaluation of *BinAuthor*

Source	# of authors	# of programs	# of functions	average # of code lines	# of files
GitHub	5	10	40000	5000	754
Google Code Jam	50	250	10050	80	250
Planet Source Code	44	168	11650	250	400
Graduate Student Projects	25	125	9609	250	450

5.4.3 Dataset Compilation

We compile the source code with different compilers and compilation settings to measure the effects of such variations. We use GNU Compiler Collection's gcc or g++ with different optimization levels, as well as Microsoft Visual Studio (VS) 2010. We study the impact of Clang and ICC compilers, as described in Section 5.4.8.

5.4.4 Author Classification

After extracting features, we define a probability value *P(A)* based on obtained weights as described in Section 5.3.3. Further, a decision function ascribes an $author_{ID}$ to any new program based on a given set of known authors. The attribution probability is defined as follows:

$$P(A) = \sum_{i=1}^{5} w_i * \lambda_i$$

where w_i represents the weight assigned to each choice, as shown in Table 5.4, and λ_i is the distance metric value obtained from each choice ($\lambda_g, \lambda_v, \lambda_q, \lambda_e$, and λ_s). If *P(A)*$\geq \zeta$, where ζ represents predefined threshold values, it is labeled as a matched author. Through our experiments, we find that the best value of ζ is *0.87*.

5.4.5 Accuracy

The main purpose of this experiment is to evaluate the accuracy of author identification in binaries.

Evaluation Settings: The evaluation of *BinAuthor* system is conducted using the datasets described in Section 5.4.2. The data is randomly split into 10 sets, where one set is reserved as a testing set, and the remaining sets are used as training sets to evaluate the system. To evaluate *BinAuthor* and to compare it with existing methods, precision (*P*) and recall (*R*) measures are applied. We choose $F_{0.5}$ because *BinAuthor* is much more sensitive to false positives than false negatives. Therefore, precision is of higher priority than recall. We employ an F-measure as follows:

$$F_{0.5} = 1.25 \, . \, \frac{P \, . \, R}{0.25P + R}$$

Results Comparison. We compare *BinAuthor* with the existing authorship attribution methods [27, 43, 112]. We evaluate the authorship classification technique presented by Rosenblum et al. [112], whose source code is available at [9], although the dataset is not

available. The source code of the proposed technique by Caliskan-Islam et al. [43] is available at [13]. For our previous system (OBA2) [27], we have the source code as well as the dataset. Caliskan-Islam et al. present the largest scale evaluation of binary authorship attribution in related work, which contains *600* authors with *8* training programs per author. Rosenblum et al. present a large-scale evaluation of *190* authors with at least *8* training programs, while Alrabaee et al. present a small scale evaluation of 5 authors with *10* programs for each. As the datasets are not available, we compare our results with these methods by using the datasets mentioned in Table 5.5. The system of Caliskan-Islam et al. uses *4500* features; Rosenblum et al. use *10000* features; Alrabaee et al. use *6500* features; and our system uses *2200* features.

Figure 5.4 details the results of comparing the accuracy between *BinAuthor* and all other existing methods. It shows the relationship between the accuracy ($F_{0.5}$) and the number of authors present in all datasets, where the accuracy decreases as the size of author population increases. The results show that *BinAuthor* achieves better accuracy in determining the author of binaries. Taking all four approaches into consideration, the highest accuracy of authorship attribution is close to *96%* on the Google Code Jam with less than *50* authors, while the lowest accuracy is *22%* when *150* authors are involved on all dataset together. We believe that the reason behind Caliskan-Islam et al. approach superiority on Google Jam Code is that this dataset is simple and can be easily decompiled to source code. *BinAuthor* also identifies the author of Github dataset with an accuracy of *90%*. The main reason for this is due to the fact that the authors of projects in Github

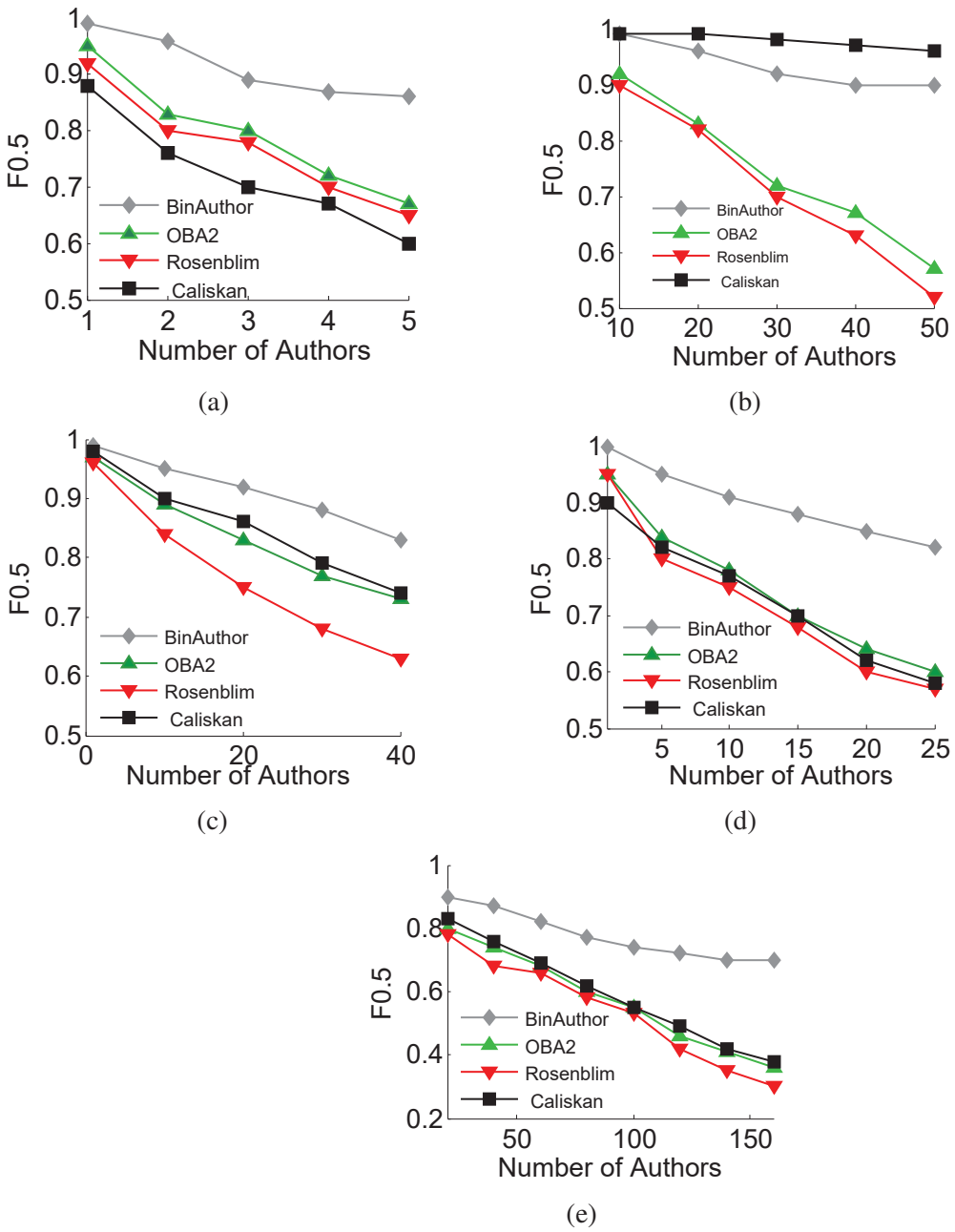


Figure 5.4: Accuracy results of authorship attribution obtained by *BinAuthor*, Caliskan-Islam et al. [43], Rosenblum et al. [112], and OBA2 [27], on (a) Github, (b) Google Code Jam, (c) Planet Source Code, (d) Graduate Student Projects, and (e) All datasets.

have no restrictions when developing projects. In addition, the advanced programmers of such projects usually design their own class or template to be used in the projects. The lower accuracy obtained by *BinAuthor* is approximately *75%* on a Graduate student projects with *25* authors. This is explained by the fact that programs in Graduate student projects have common choices among different students due to assignment rules, which force students to change/restrict their habits accordingly. When the number of authors is *140* on the mixed dataset, the accuracy of Rosenblum et al., Caliskan-Islam et al., and OBA2 approaches drop rapidly to *30%* on all datasets, whereas our system's accuracy remains greater than *75%*. This provides evidence for the stability of using coding habits in identifying authors. In total, the different categories of choices achieve an average accuracy of *84%* for ten distinct authors and *75%* when discriminating among *152* authors. These results show that author habits may survive the compilation process.

5.4.6 False Positives

We investigate the false positives in order to understand the situations where *BinAuthor* is likely to make incorrect attribution decisions. Figure 5.5 shows the false positives relationship with the number of authors in repository. For this experiment, we consider 4 programs for each author. For instance, when we have 50 authors (4*50 = 200 programs), *BinAuthor* misclassifies 16 programs. Also, when the number of authors is 1100 (1100*4 = 4400 programs), the number of false positives is 402. These false positives (402) are investigated in Figure 5.5 (a). Also, we show the false positives in each dataset as shown

in Figure 5.5 (b). It is obviously shown that the false positives rate for student dataset is the highest rate and we believe the reason behind this is that each student should follow the standard coding instructions that restrict him/her to have their habits.

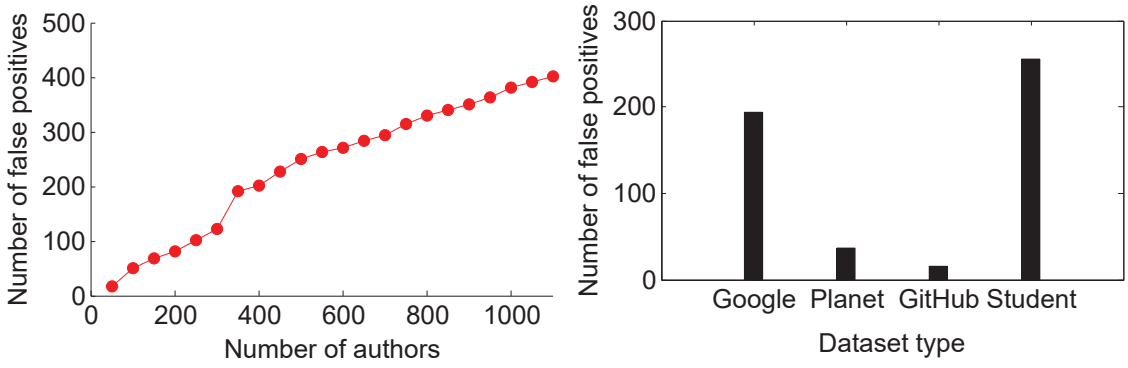


Figure 5.5: False positive analysis.

5.4.7 Scalability

Security analysts or reverse engineers may be interested in performing large-scale author identification, and in the case of malware, an analyst may have to deal with a large number of new samples on a daily basis. With this in mind, we evaluate how well *BinAuthor* scales. To prepare a large dataset for the purpose of large-scale authorship attribution, we obtained programs from three sources: Google Code Jam, GitHub, and Planet Source Code. We eliminated from the experiment programs that could not be compiled because they contain bugs and those written by authors who contributed only one or two programs. The resulting dataset comprised 103,800 programs by 23,000 authors: 60% from Google

Code Jam, 25% from Planet source code, and 15% from GitHub. We modified the script[1] used in [43] to download all the code submitted to the Google Code Jam competition. The programs from the other two sources were downloaded manually. The programs were compiled with the Visual Studio and gcc compilers, using the same settings as those in our previous investigations. The experiment evaluated how well the top-weighted choices represent author habits. The results of the large-scale author identification are shown in Figure 5.6.

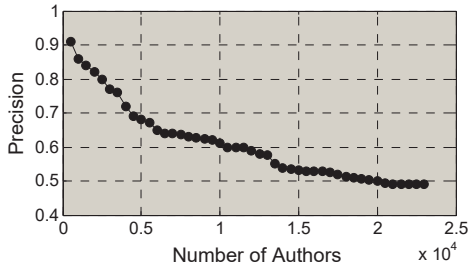


Figure 5.6: Large-scale author attribution

The figure shows the precision with which *BinAuthor* identifies the author, and its scaling behavior as the number of authors increases is satisfactory . An author is identified among almost 4000 authors with 72% precision. When the number of authors is doubled to 8000, the precision is close to 65%, and it remains nearly constant (49%) after the number of authors reaches $19,000$. Additionally, we tested *BinAuthor* on the programs from each of the sources. We found high precision (88%) for samples from the GitHub dataset, 82% precision for samples from the Planet dataset, and low precision (51%) for samples from Google code jam. After analyzing these results, we find that the authors

[1] https://github.com/calaylin/CodeStylometry/tree/master/Corpus

who wrote the code for difficult tasks is easier to attribute than easier tasks.

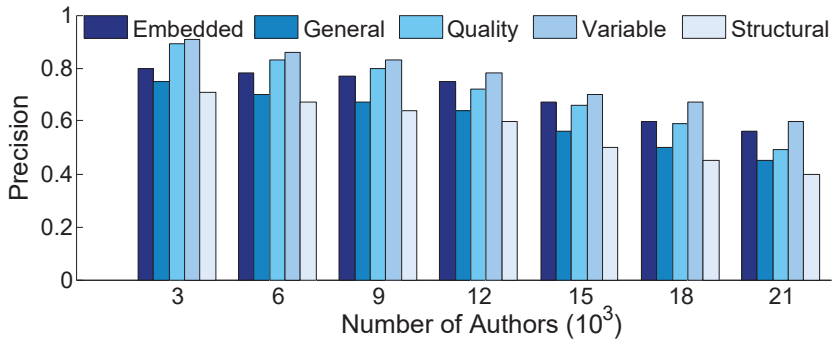


Figure 5.7: Effect of choices on large-scale author identification

We studied the impact of each choice on precision (Figure 5.7). For example, when the number of authors is 15,000, *BinAuthor* achieves a precision of 70% based on the use of variables, while with structural considerations, it achieves a precision of 50%, the lowest for all the choices. When the number of authors reaches 21,000, the precision for embedded, general, quality, variable, and structural choices is 56%, 45%, 49%, 60%, and 40%, respectively. From Figure 5.6, it seems reasonable to expect that when the number of authors exceeds 20,000, there will be little additional change in the precision.

5.4.8 Impact of Evading Techniques

Refactoring Techniques. We consider a random set of 50 files from our dataset which we use for the C++ refactoring process [1, 16]. We consider the techniques described in 2.3. We obtain an accuracy of 83.5% in correctly classifying authors, which is only a mild drop in comparison to the 85% accuracy observed without applying refactoring techniques. Based on the above results, *BinAuthor* can tolerate refactoring techniques, as some of

Table 5.6: Evading techniques: methods used, tools used, and their affect on *BinAuthor* choices.

Tools	Method	Input	Output	A $\longrightarrow A^*$	Choice				
					General	Variable	Quality	Embedded	Structural
Refactoring [1, 16]	RV	Binary	Binary	85 $\longrightarrow$ 84	○	●	○	○	○
	MM			86 $\longrightarrow$ 82	○	○	○	●	●
	NM			86 $\longrightarrow$ 83	○	○	○	○	○
Obfuscator-LLVM [77]	CFG flattening	Binary	Binary	86 $\longrightarrow$ 83	○	○	○	○	●
	Instruction substitution			86 $\longrightarrow$ 80	○	●	●	○	○
	CFG bogus			86 $\longrightarrow$ 81	○	○	○	○	●
DaLin [87]	Instruction reordering	Assembly	Assembly	86 $\longrightarrow$ 86	○	○	○	○	○
	Dead code insertion			86 $\longrightarrow$ 86	○	○	○	○	○
	Register renaming			86 $\longrightarrow$ 86	○	○	○	○	○
	Instruction replacement			86 $\longrightarrow$ 80	○	●	●	○	○
Trigress [25]	Virtualization	Source	Source	86 $\longrightarrow$ 20	○	○	○	○	○
	Jitting			86 $\longrightarrow$ 0	○	○	○	○	●
	Dynamic			86 $\longrightarrow$ 10	●	○	○	○	●
PElock [23]	Hide procedure call	Assembly	Assembly	86 $\longrightarrow$ 85	○	○	○	○	●
	Insert fake instruction			86 $\longrightarrow$ 86	○	○	○	○	○
	Prefix junk opcode			86 $\longrightarrow$ 86	○	○	○	○	○
	Insert junk handlers			86 $\longrightarrow$ 86	○	○	○	○	○
Nynaeve [21]	Frame Pointer Omission	Source	Binary	86 $\longrightarrow$ 83	●	○	○	○	○
	Function inlining			86 $\longrightarrow$ 78	○	●	●	○	●
OREANS [22]	Encrypt binary	Source	Binary	NA	NA	NA	NA	NA	NA
Gas Obfuscator [17]	Junk byte	Assembly	Assembly	86 $\longrightarrow$ 86	○	○	○	○	○
Designed Script	Loop unrolling	Source	Source	86 $\longrightarrow$ 75	●	●	●	○	○

Note: (A) means the accuracy before applying obfuscation method while (A^*) means the accuracy after applying it. (○) means there is no effect while (●) means there is an effect. (*NA*) means is not applicable.

these techniques change the syntax of the code but do not change its semantics. The accuracy remains the same when the RV technique is applied, whereas the accuracy drops slightly when MM and NM are applied. Since some of the choices used in *BinAuthor* (general and embedded choices) are based on semantic features, they are not significantly affected by these techniques. However, qualitative choices are more affected since these choices rely on specific patterns captured/represented by idioms.

Impact of Obfuscation. We are interested in determining how *BinAuthor* handles simple binary obfuscation techniques intended for evading detection, as implemented by tools such as Obfuscator-LLVM [78]. These obfuscators replace instructions by other semantically equivalent instructions, introduce spurious control flow, and can even completely flatten control flow graphs. For this experiment, we consider a set of 50 authors from our dataset, all of whom have five binary samples. Obfuscation techniques implemented by Obfuscator-LLVM are applied to the samples prior to classifying the authors. We proceed to extract functionality-independent choices from obfuscated samples. Using principle component analysis to select the best features, we obtain an accuracy of 82.9% in correctly classifying authors, which is only a slight drop in comparison to the 85% accuracy observed without obfuscation. We combine the refactoring process with the above obfuscation by first applying the refactoring techniques on the selected dataset (50 authors) at the source level, after which they are compiled using Visual Studio 2010. After applying obfuscation techniques, the accuracy dropped from 85% to 80.4%. Table 5.6 shows the details about which tool has been used and which methods are applied. In addition, we

show which choice in *BinAuthor* is affected.

The Impact of Compilers. To create experimental datasets for this purpose, we consider *1000* authors with five training programs for each. We first compile the source code with gcc, VS, ICC, and Clang compilers. Next, to measure the effect of different compilation options such as compiler optimization flags, we additionally compile the source code with level-1, level-2, and level-3 optimizations, namely the Od, O2, and Ox flags. The results show that for most optimization speed levels coding habits are preserved to a great extent. However, the accuracy drops significantly more (from *86%* to *43%*) when the Clang or ICC compilers are used compared to the slight drop in accuracy (from *86%* to *83%*) when the VS and gcc compilers are used, as the former compilers produce more variable code.

5.5 Applying *BinAuthor* to Malware Binaries

One challenge in applying *BinAuthor* to real world malware is the lack of ground truth concerning the attribution of authorship due to the nature of malware. Also, whether a malware package is created by an individual or an organization is generally unconfirmed. Those limitations partially explain the fact that few research efforts have been seen on this subject. In fact, to the best of our knowledge, *BinAuthor* is the first attempt to apply automated authorship attribution to real malware. Fortunately, we can correlate the results of our automated approach to those obtained through manual analysis by domain experts, e.g., in [91], a manual investigation is conducted to establish relationships among

the authors of a few samples of malware. We have applied *BinAuthor* to a set that comprises three pairs, each suspected of having common authorship: `Bunny` and `Babar`; `Stuxnet` and `Flame`; and `Zeus` and `Citadel`. In [10, 85, 91], it is found that each of the pairs in the dataset is likely to have originated from the same set of authors. Table 5.7 describes the characteristics of the first malware dataset.

Table 5.7: Characteristics of malware datasets

Malware	Packed	Obfuscated	Source code	Binary code	Type	# of functions	Source of sample
`Zeus`	✗	✗	✓	✓	PE	557	Contagio [2]
`Citadel`	✗	✗	✓	✓	PE	794	Contagio [2]
`Flame`	✗	✓	✗	✓	ELF	1434	Contagio [2]
`Stuxnet`	✗	✓	✗	✓	ELF	2154	Contagio [2]
`Bunny`	✓	✗	✗	✓	PE	854	VirusSign [4]
`Babar`	✓	✗	✗	✓	PE	1025	VirusSign [4]

We describe the application of *BinAuthor* to some well-known malware binaries. Given a set of functions, *BinAuthor* clusters them based on the number of common choices. The existence of three or more shared choices is an indication that the functions are likely to have a single author. Sharing only one or two choices suggests multiple authors due to the lack of stylistic consistency.

Table 5.8: Statistics of applying *BinAuthor* to malware binaries

Malware	Number of functions with common choices					Number of common functions with			
	General	Qualitative	Structural	Embedded	Variable	1 choice	2 choices	3 choices	4 choices
`Bunny` and `Babar`	372	494	127	450	278	290	150	478	340
`Stuxnet` and `Flame`	725	528	189	300	0	689	515	294	180
`Zeus` and `Citadel`	655	452	289	370	0	600	588	194	258

5.5.1 Applying *BinAuthor* to `Bunny` and `Babar`

Findings. We apply *BinAuthor* to binaries and cluster the functions based on functionality independent choices. *BinAuthor* is able to find the following coding habits automatically: the use of all capital letters for *config* in XML and the preference for using Visual Studio 2008 (general choices); the use of one variable over a long chain (variable choice); the choice of methods for accessing freed memory, dynamically deallocating allocated resources, and reopening resources more than once in the same function (quality choices); and the use of a common approach to managing functions (structural choices).

Statistics. As shown in Table 5.8, *BinAuthor* found functions common to `Bunny` and `Babar` that share the aforementioned coding habits: *494* functions share qualitative choices; *450* functions share embedded choices; *372* functions share general choices; *278* functions share variable choices; and *127* functions share structural choices. Among these, *BinAuthor* found *340* functions that share *4* choices, *478* functions that share *3* choices, *150* functions that share *2* choices, and *290* functions that share *1* choice.

Summary. Considering the 854 and 1025 functions in `Bunny` and `Babar`, respectively, BinAuthor found that 44% ((340 + 478) / (854 + 1025)) are likely to have been written by a single author, and 23% are likely to have been written by multiple authors. No common choices were identified in the remaining 33%, likely because different segments or code lines within the same function were written by different authors, a common practice in writing complex software.

5.5.2 Applying *BinAuthor* to `Stuxnet` and `Flame`

Findings. *BinAuthor* found the following coding habits automatically: the use of global variables, `Lua` scripting language, a specific open-source package `SQLite`, and heap sort rather than other sorting methods (general choices); the choice of opening and terminating processes (qualitative choices); the presence of recursion patterns and the use of POSIX socket API rather than BSD socket API (structural choices); and the use of functions that are close in terms of the Mahalanobis distance, with distance close to 0.1; and the passing of primitive types by value, but the passing of objects by reference (embedded choices).

Statistics. As shown in Table 5.8, *BinAuthor* identified functions common to `Stuxnet` and `Flame` that share the aforementioned coding habits.

Summary. *BinAuthor* clustered the functions and found that *13% ((180 + 294) / (1434 + 2154))* were written by one author, while *34% ((515 + 689) / (1434 + 2154))* were written by multiple authors. No common choices were found in the remaining *53%* of the functions. The fact that these malware packages follow the same rules and set the same targets suggests that `Stuxnet` and `Flame` are written by an organization.

5.5.3 Applying *BinAuthor* to `Zeus` and `Citadel`

Findings. *BinAuthor* identified the following coding habits: the use of network resources rather than file resources, creating configurations using mostly config files, the use of specific packages such as `webph` and `ultraVNC` (general choices); the use of `switch` statements rather than `if` statements (structural choices); the use of semaphores and locks

(qualitative choices); and the presence of functions that are close in terms of the Mahalanobis distance, with distance = 0.0004 (embedded choices).

Statistics. As listed in Table 5.8, *BinAuthor* found functions common to `Zeus` and `Citadel` that share the aforementioned coding habits.

Summary. After *BinAuthor* clustered the functions, it appears that 33% were written by a single author, while *53%* were written by the same team of multiple authors. No common choices were found for the remaining 14% of the functions. Our findings clearly support the common belief that `Zeus` and `Citadel` were written by the same team of authors.

5.5.4 Verifying correctness of *BinAuthor* Findings

Due to the lack of ground truth, we verify the correctness of *BinAuthor* findings using following methods: Comparing BinAuthor outputs to the findings of human experts in available technical reports [10, 85, 91]; measuring the distance between the choices in one cluster and the choices in another to calculate the degree of similarity; measuring the degree of similarity between the extracted choices from the two malware packages in one pair and those from the second dataset (for which we have the ground truth since the source code is available) to provide a clear indication of whether the choices are closely related to specific malware packages.

Comparison with technical reports. We compare the *BinAuthor* findings with those made by human experts in technical reports. For `Bunny` and `Babar`, our results match

the technical report published by the Citizen Lab [91], which demonstrates that both malware packages were written by a set of authors according to common implementation traits (general and qualitative choices) and infrastructure usage (general choices). The correspondence between the *BinAuthor* findings and those in the technical report is the following: 60% of the choices matched those mentioned in the report, and 40% did not; 10% of the choices found in the technical report were not flagged by *BinAuthor* as they require dynamic extraction of features, while *BinAuthor* uses a static process.

For `Stuxnet` and `Flame`, our results corroborate the technical report published by Kaspersky [85], which shows that both malware packages use similar infrastructure (e.g., Lua) and are associated with an organization. In addition, the *BinAuthor* findings suggest that both malware packages originated from the same organization. The frequent use of particular qualitative choices, such as the way the code is secured, indicates the use of certain programming standards and strict adherence to the same rules. Moreover, the *BinAuthor* findings provide much more information concerning the authorship of these malware packages. The correspondence between the *BinAuthor* findings and those in the technical report is as follows: all the choices found in the report [85] were found by *BinAuthor*, but they represent only 10% of our findings. The remaining 90% of the *BinAuthor* findings were not flagged by the report.

For `Zeus` and `Citadel`, our results match the findings of the technical report published by McAfee [10], indicating that `Zeus` and `Citadel` were written by the same team of authors. The correspondence between the findings of *BinAuthor* and those of

McAfee are as follows: 45% of the choices matched those in the report, while 55% did not, and 8% of the technical report findings were not flagged by *BinAuthor*.

Measuring similarity between choices in malware binaries. In this section, the goal is to assess the similarity between malware binaries by reporting the statistics about common choices (Table 5.9). We observed that there are only ten choices common to `Bunny` and `Stuxnet`, which clearly indicates that the malware packages were written by different authors. These choices are found in seven functions, which amounts to *(7/(854 + 2154))= 0.2%* shared author habits. In comparison, there are seventeen choices common to `Flame` and `Zeus`, found in thirty-eight functions, so the percentage of shared author habits is *(38 / (1434 +557)) = 2%*. The results in Table 5.9 may provide clues about the validity of the *BinAuthor* findings.

Table 5.9: Choices found in malware binaries

	Bunny	Babar	Stuxnet	Flame	Zeus	Citadel
Bunny	-	500	10	2	4	12
Babar	500	-	4	9	0	5
Stuxnet	10	4	-	750	14	3
Flame	2	9	750	-	17	6
Zeus	4	0	14	17	-	670
Citadel	12	5	3	6	670	-

Measuring the degree of similarity between ground truth datasets and malware binaries.

As another verification of the correctness of the findings, we measured the degree of similarity between the dataset used here and other datasets for which we have the ground truth (e.g., Google code jam) to see how likely such a degree of similarity could come

from shared authorship. The goal of computing the degree of similarity is to determine whether the habits found in the malware binaries are present to the same degree in conventional binaries, which will reveal whether these habits are indeed specific to malware writers. To provide an even more convincing verification, we computed the similarity scores between related pairs of malware and the rest of the available dataset. The results are presented in Table 5.10. *BinAuthor* found a total of 500 choices in `Bunny` & `Babar`, of which 45 choices, i.e., only 1%, are similar to those in the Student project dataset. We believe that one of the main reasons for the low similarity is that the programmers participating in the Google code jam may have greater expertise, more extensive background knowledge, and better skills than the typical malware writer. Another comparison revealed that *104/500 = 21%* of the choices are common to GitHub authors and malware writers. At the same time, the choices in `Stuxnet` & `Flame` have less similarity with the other datasets: 2%, 1%, 0.2%, and 4% for Google code jam, planet code, student code, and GitHub code, respectively.

Table 5.10: Number of choices common to the malware dataset and the ground truth dataset

	Google	Planet	Student	GitHub
`Bunny` & `Babar`	6%	6%	1%	10%
`Stuxnet` & `Flame`	1%	4%	0%	7%
`Zeus` & `Citadel`	7%	9%	3%	17%

5.6 Summary

To conclude, we have presented the first known effort on decoupling coding habits from functionality. Previous research has applied machine learning techniques to extract stylometry styles and can distinguish between 5-50 authors, whereas we can handle up to 1500 authors. In addition, existing works have only employed artificial datasets, whereas we included more realistic datasets. In summary, our system demonstrates superior results on more realistic datasets.

Chapter 6

Towards Extracting Semantics of Binary Code

6.1 Overview

In this chapter, we propose a novel technique that extracts the semantics of binary code in terms of both data and control flow. Our technique allows more robust binary analysis because the extracted semantics of the binary code is generally immune from code transformation techniques and varying the compilers or compilation settings. Specifically, we apply data-flow analysis to extract the semantic flow of the registers as well as the semantic components of the control flow graph, which are then synthesized into a novel representation called the semantic flow graph (SFG). Subsequently, various properties, such as reflexive, symmetric, antisymmetric, and transitive relations, are extracted from

the SFG and applied to binary analysis. We implement our system in a tool called *BinGold* and evaluate it against thirty binary code applications. Our evaluation shows that *BinGold* successfully determines the similarity between binaries, yielding results that are highly robust against code transformation techniques. In addition, we demonstrate the application of *BinGold* to two important binary analysis tasks: binary code authorship attribution, and the detection of reused functions across program executables. The promising results suggest that *BinGold* can be used to enhance existing techniques, making them more robust and practical.

6.2 Motivating Example

We start with a simple example composed of part of MD5 written in C++ (Listing 1). In this sample, the hex representation of the digest is returned as a string. MD5 performs many binary operations on the "message" (text or binary data) to compute a 128-bit "hash". We compile this part of the MD5 example code on Windows 7 using g++, Visual Studio 2010, Clang, and ICC. We then use IDA to disassemble the binary. Many security tools use IDA in this way, as a first step before performing additional analysis [71, 104].

Listing 6.1: Motivating example: Part of MD5 method

```
std::string MD5::hexdigest() const {
  if (!finalized)
    return "";
```

```
    char buf[33];
    for (int i=0; i<16; i++)
      sprintf(buf+i*2, "%02x", digest[i]);
    buf[32]=0;

    return std::string(buf);
}
```

We compute the control flow graph for the fragment and then compare them as illustrated in Table 6.1. We notice through the motivating example that the compiler also makes changes to both the control structure and the basic blocks and hence instructions. We show a list of traditional features in Table 6.2.

Table 6.1: Graph features applied on CFGs for the fragment code in Listing 1, which is compiled by visual studio, ICC, g++, and Clang

Feature	Graph A	Graph B	Graph C	Graph D
# of nodes	8	8	13	5
# of edges	9	8	15	4
K-cone	0-4	0-6	0-4	0-3
Radius	2	3	5	2
Width of graph	3	2	4	2
Length of graph	5	7	5	4
Diameter	3	4	6	2
Cyclometry Complexity	3	2	4	1

We name the graphs as graph A, graph B, graph C, and Graph D; these graphs represent CFGs from visual studio, ICC, g++, and Clang, respectively. We can see in Table 6.1 that among some graphs, there are features with the same values; for example the number of nodes is the same for graphs A and B. Cyclomatic complexity varies; it is calculated by $M = E - N + 2P$, where E is the number of edges, N is the number of nodes,

and P is the number of connected components. Additionally, we observe there are some common values between graphs A and C. For instance, the number of nodes is 8 when it is compiled with Visual Studio, but it is 13 with g++ and 5 with Clang. Additionally, the number of edges ranges from 4 to 17.

Table 6.2: Graph features description

Feature	Description
Number of nodes	Number of basic blocks
Number of edges	Number of control flows (i.e., true)
K-cone	K represents the number of CFG level
Radius	Minimum vertex eccentricity
Width of graph	Maximum number of nodes at the same level
Length of graph	Number of nodes in the longest path
Diameter	The longest shortest path between any two nodes in the graph
Cyclometry Complexity	Number of linearly independent paths within the CFG

As a result of the aforementioned differences, the structural approaches may lead to false positives by claiming that two graphs are the same (because of similar graph features), when in fact they are not. Additionally, we observe through the motivating example that there are differences in instructions at the syntax level; these differences affect the results of the syntax approaches in terms of reporting similarities. Hence, the necessity of having an automated tool that can simply extract the semantics of a code will significantly reduce the percentage of false positives.

6.3 Extracting Semantics of Binary Code

In this section, we describe how we built upon the background in Section 2 to perform the task of extracting the semantics of a binary code.

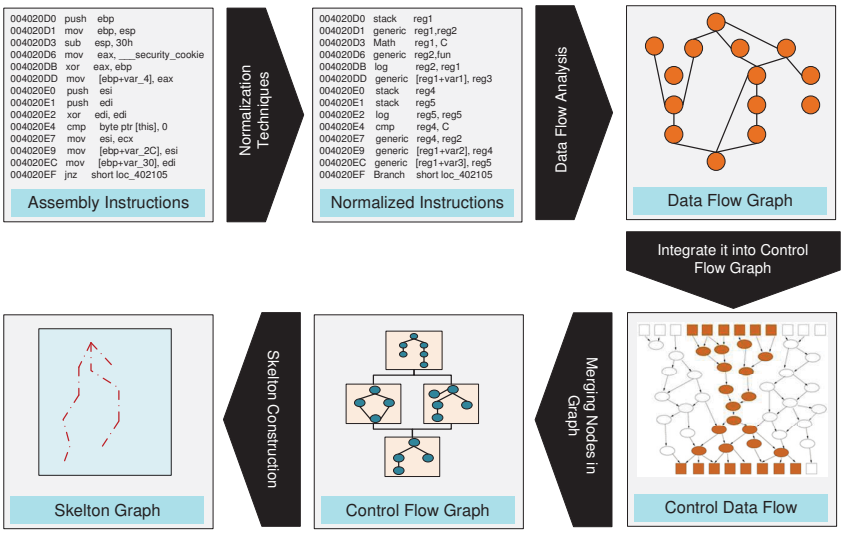


Figure 6.1: Architecture overview

6.3.1 Architecture Overview

Our architecture employs a series of techniques illustrated in Figure 6.1 and described in the upcoming sections. First, the binary code is disassembled by IDA Pro [8] disassembler. Second, a set of rules are applied to assembly instructions to normalize the code. Third, data flow rules are applied to these normalized instructions to construct data flow dependencies. In addition, we extract the semantics of the CFG by constructing a conservative approximation of the target function prototype by means of a use-def analysis of possible callees. We then couple these results with liveness analysis at each indirect call site to arrive at a many-to-many relationship between call sites and target callees in order to recover call site and callee signatures. Both types of semantics are integrated into a new representation called the SFG. Subsequently, the properties of the SFG, such as the reflexive, symmetric, and transitive relations are extracted from the SFG.

6.3.2 Data Flow Graph Construction

After normalizing the instructions, we apply data flow to infer the program variable relations using coarse reasoning about the program control flow and data dependencies. Depending on how such analyses choose to model the flow of information through the data structures. Let R_k/W_k denote registers or memory that instruction I_k reads or writes. If i_1 and i_2 are instructions belonging to I and they are in the same basic block, then we define the following possible dependencies: i_1 writes something which will be read by i_2; i_1 reads something before i_2 overwrites it; and i_1 and i_2 both write the same variable. This category of dependency is considered an internal dependency. The other dependency is control dependence. If i_1 and i_2 are both in the same basic block, and i_2 is a control instruction, we call it an internal control dependence. Also, i_1 and i_2 are in two different basic blocks, where i_1 is the last instruction in the first basic block and i_2 is executed in the second basic block as the first instruction, where the second basic block is a successor of the first basic block in the control flow graph, then it is also an internal control dependence.

6.3.3 Equivalence Relations and Partitions in SFG

The data flow graph together with the invariants form the semantic flow graph. We combine this semantic information to form a new representation in order to facilitate more efficient graph matching between different binary codes for determining the similarity or integrating into some existing frameworks. Formally, a semantic flow graph (SFG) is

defined as follows.

Definition 3. A semantic flow graph G = (N,V, ζ, γ, ϑ, λ, ω) is a directed attributed graph where N is a set of nodes, $V \subseteq (N \times N)$ is a set of edges and ζ is edge labeling function which assigns a label to each edge: $\zeta \longrightarrow \gamma$, where γ is a set of labels (internal dependency or external dependency). ϑ is a call-callee relation function which colors each node $n \epsilon N$ based on its relation with other node $k \epsilon N$. Finally, ω is a function for coloring dataflow control or data dependencies.

We illustrate a simple example in Figure 6.2 to show how SFG could be constructed. As shown, ω is a function for coloring dataflow dependencies; control or data dependency. ϑ is a call-callee relation function. We can notice the green color in Figure 6.2 (c) represents caller-callee relation. For instance, *i2* has a caller-callee relation with *i5*. Besides,

We then construct the relations from the SFG. We generalize equivalence relations and equivalence classes, where an equivalence relation on a set of features (semantics features) *F* is a relation $R \subset F\ x\ F$ such that:

- $(f_i, f_j)\ \varepsilon\ R$ for all $f \varepsilon F$, which is called the reflexive property
- $(f_i, f_j)\ \varepsilon\ R$ implies $(f_j, f_i)\ \varepsilon\ R$
- (f_i, f_j) and $(f_j, f_k)\ \varepsilon\ R$ imply (f_i, f_k)

We also extract a collection of nonempty sets of features *F*, which is called partition *P*. This is a collection of nonempty sets $f_1, f_2, ...$ such that $f_i \bigcap f_j = for\ i \neq j$ and

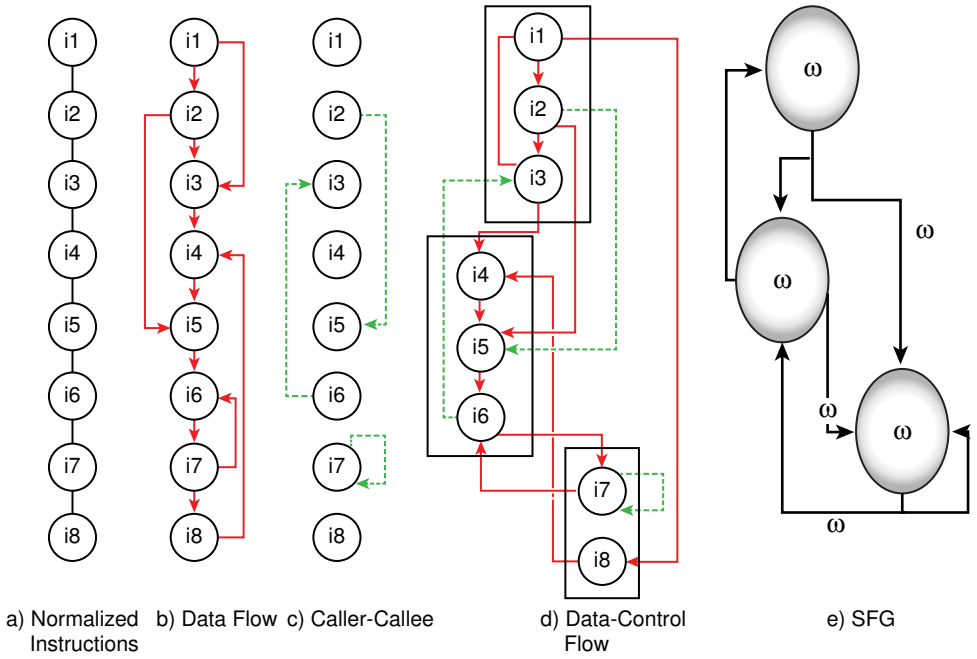


Figure 6.2: Example of constructing SFG

$\bigcup_k F_k = X$. Let $\sim$ be an equivalence relation on a set F and let $f \varepsilon F$. Then $[f] = \{f_j \varepsilon F : f_j \sim f_i$ is called the equivalence class of $f\}$.

6.4 Detection Process

We next describe the detection system *Bingold*. Since *Bingold* extracts different types of features that capture the semantics of code, the detection system is composed of multiple components employing a series of techniques, as depicted in Figure 6.3 and explained in the next subsections.

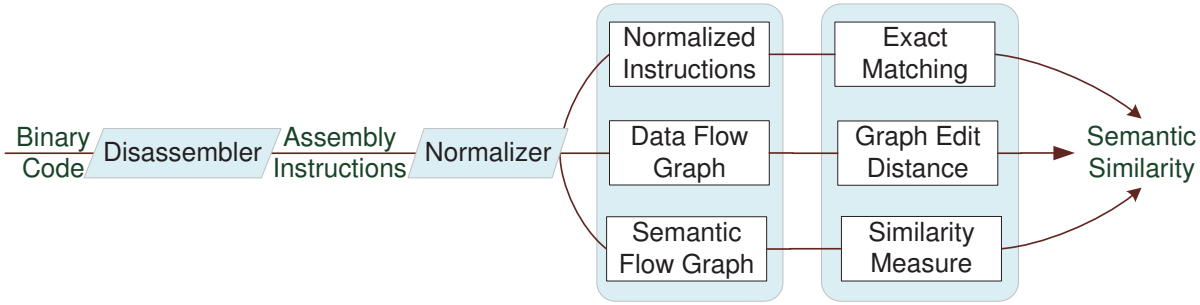


Figure 6.3: Detection system

6.4.1 Exact Matching

As previously described, we normalize the code according to predefined rules and then apply the predefined categories to those normalized instructions. We then convert those instructions to hash vectors. Finally, we match instructions together.

6.4.2 Graph Edit Distance

For inexact matching between data flow graphs, a distance metric is needed. We employ Algorithm 1 introduced in 3.3.4. Given two data flow graphs, to transform one graph into another, we define two concepts: `internal flow dependency` and `external flow dependency`. The edit distance between two data flow graphs *G* and *H* is thus defined as the minimum weight of all dependencies *d* between them; i.e., $sim(G, H) = min\ w(V_{G,H})$, where V is the function for checking the dependencies. We also use the same dissimilarity introduced in 3.3.4 between two data flow graphs G and H.

6.4.3 Similarity Measure

For the extracted relations, we compare two graphs in terms of the similarity of their reflexive, symmetric, and transitive relations. Given two data SFGs G and H, we define the similarity measure $sim(G, H) = max\ R(V_{G,H})$. ***R*** is a function extracts the common relations between two graphs and measures the similarity between them.

6.4.4 Weight Parameter Settings

We define for each component (data flow, caller-callee relationship, and SFG) in our system a weight. These weights are: α, β, and γ, to determine the contribution of each component. We experimentally determine the optimal values for these parameters. The parameter setting is computed using nine-fold cross-validation. We evaluate values of α ranging from 0 to 1 in steps of size 0.1 and β ranging from 0 to 1 in steps of size 0.1. For a given choice of α, β, and γ, it is required that α + β + γ = 1. In each setting, the features are extracted using our system and the F_1 score is computed that the maximum F_1 score is obtained for α = 0.5, β = 0.2, and γ = 0.3. We use these values as the default for $BinGold$ as well as throughout the rest of the evaluation.

6.5 Evaluation

This section details the evaluation of our system. Section 6.5.1 describes the dataset used in our evaluation. Section 6.5.2 presents the evaluation metrics. Section 6.5.3 shows

the results of our system for different compilers and compilation settings. Section 6.5.4 shows the robustness of our system against code transformation techniques. Finally, section 6.5.6 shows the effect of integrating our system into certain existing approaches and demonstrates improvements in accuracy.

6.5.1 Dataset

We evaluate our system against 30 programs for which we have the source code. These programs are only used to extract the ground truth by compiling the source code with debugging information.

Table 6.3 summarizes the 30 programs. For each program, the table shows the program identifier, the program name, the binary code statistics, and the source compiler. From the binary code it captures the type of executable generated (PE or ELF) and the number of functions in the executable. The binary code information is extracted using IDA pro [8] by reading the executable's debugging information. 3 projects compiled by 4 compilers, 8 projects compiled by 3 compilers, and 19 projects compiled by 2 compilers. The dependency of the program restricts us to compiling each project using 4 compilers.

Our dataset are open-source projects from SourceForge [24], and the GNU software repository [18]. Our dataset includes 17 PE binaries and 13 ELF binaries. We include multiple programs from the same project that could be compiled by different compilers and use those programs to analyze the applicability and efficiency of our system.

Table 6.3: Programs used in our system evaluation

		Binary Code		
ID	Program	Type	Function	Compiler
1	SQlite	PE	3920	VS, GCC, ICC, Clang
2	OpenSSL	PE	2163	VS, GCC
3	info-zip	PE	1784	VS, ICC
4	jabber	PE	5910	VS, GCC
5	Hashdeep	PE	2905	VS, Clang, GCC
6	libpng	PE	9226	VS, GCC
7	ultraVNC	PE	3526	VS, GCC
8	lcms	PE	1082	Clang, ICC, GCC
9	ibavcodec	PE	739	VS, GCC, ICC
10	TrueCrypt	PE	1093	VS, GCC
11	libjsoncpp	PE	4114	VS, ICC
12	7z	PE	2179	VS, GCC, ICC
13	7zG	PE	2530	VS, GCC, ICC
14	7zFM	PE	3149	VS, GCC, ICC
15	lzip	ELF	33	VS, GCC
16	tinyXMLTest	ELF	2744	VS, GCC, ICC, Clang
17	libxml2	ELF	58	VS, GCC, ICC
18	Mersenne Twister	ELF	2740	VS, GCC
19	bzip2	ELF	285	VS, GCC
20	lshw	ELF	1429	VS, GCC
21	smartctl	ELF	457	VS, GCC
22	pdftohtml	ELF	499	VS, GCC, Clang
23	ELF statifier	ELF	2340	VS, GCC
24	FileZilla	PE	6250	VS, GCC
25	ncat	PE	1855	VS, GCC
26	Hasher	PE	436	VS, GCC, ICC, Clang
27	tfshark	ELF	439	VS, GCC
28	dumpcap	ELF	448	VS, GCC
29	tshark	ELF	1008	VS, GCC
30	pageant	ELF	2212	VS, GCC

6.5.2 Evaluation Metrics

To evaluate the accuracy of our system, we conducted the following experiments. First, we compared two sets of results: the results output by some existing tools (i.e., authorship attribution, clone detection) and the results after integrating our system with these tools. Second, we compared the similarity of the same program when it is compiled by different compilers and with different compilation settings. Third, we applied different code transformation techniques to the same binary file and checked the similarity based on the semantic information extracted by our system. Finally, we applied different refactoring techniques to the source code and compiled it using different compilers. We then employed our tool to measure the similarity between the binary files.

We use validity metrics such as *precision*, *recall*, and F_1. Precision *(P)* and recall *(R)* are defined as follows:

$$P = \frac{TP}{TP + FP},\ R = \frac{TP}{TP + FN} \tag{6.1}$$

where *TP* (true positives) is the number of functions assigned correctly by our system; *FP* (false positives) is the number of functions assigned incorrectly by our system; and *FN* (false negatives) is the number of functions not assigned by our system but which actually belong to it. To combine both precision and recall, we use the F_δ score with $\delta = 1$, which is equal to the harmonic mean of the precision and recall values. F_1 scores fall within the

interval $[0, 1]$, where the larger the F_1 score, the better the overall accuracy.

6.5.3 Accuracy Results of C/C++ Programs with Different Compilers and Compilation Settings

As previously mentioned, we compiled 30 programs using different compilers such as Clang and ICC. We evaluate how well our system detects the similarities among those executables using the F_1 score. Table 6.4 summarizes the results. The median F_1 score is 0.78. The precision ranges from 0.60 to 0.90, and the recall ranges from 0.64 to 0.92.

The accuracy of the C++ results is higher than the accuracy of the C results because C++ source code contains classes with small-sized methods. These small components are mostly unaffected by compilers or compilation settings. However, they may be inlined and are thus easily identified based on data flow components. For instance, the program FileZilla has the highest F_1 score of 0.90, while the program dumpcap has the lowest F_1 score of 0.63. For C programs, the median F_1 score is 0.67. The results for C binary code similarity are worse than the results for C++ programs. This is expected as C programmers are not constrained by the object-oriented paradigm and often place functions with different semantics in the same source file. For example, the file tfshark.c in tfshark combines string processing, message processing (read/write/print), and common functions for program output. These functions are technically similar in semantic representation, but the presence of all three reduces the F_1 score to 0.64 when using automated ground truth based on source files. Moreover, C programs have less modularity than C++ programs so

it may be harder to extract the semantics of a code.

Table 6.4: Our system accuracy in determining the similarity between binaries

Program	Precision	Recall	F1	Program	Precision	Recall	F1
SQlite	0.75	0.88	0.81	tinyXMLTest	072	0.79	0.75
OpenSSL	0.72	0.66	0.69	libxml2	0.78	0.82	0.80
info-zip	0.68	0.9	0.77	Mersenne Twister	0.78	0.88	0.83
jabber	0.67	0.88	0.76	bzip2	0.82	0.9	0.86
Hashdeep	0.63	0.72	0.67	lshw	0.83	0.83	0.83
libpng	0.82	0.68	0.74	smartctl	0.89	0.92	0.90
ultraVNC	0.81	0.67	0.73	pdftohtml	0.85	0.75	0.80
lcms	0.75	0.66	0.70	ELF statifier	0.83	0.74	0.78
ibavcodec	0.77	0.81	0.79	FileZilla	0.90	0.92	0.90
TrueCrypt	0.90	0.88	0.89	ncat	0.72	0.71	0.71
libjsoncpp	0.85	0.67	0.75	Hasher	0.71	0.68	0.69
7z	0.74	0.77	0.73	tfshark	0.70	0.65	0.67
7zG	0.66	0.81	0.73	dumpcap	0.62	0.64	0.63
7zFM	0.66	0.82	0.76	tshark	0.60	0.68	0.64
lzip	0.66	0.9	0.75	pageant	0.67	0.67	0.67

6.5.4 Accuracy Results after Applying Code Transformation Techniques

We consider a random set of 15 files from our dataset and compile them using Visual Studio 2010. The binaries are converted into assembly files through the disassembler, and the code is then obfuscated using the `DaLin` generator [87]. This generator applies the following: (i) register renaming (RR), which is one of the oldest and simplest techniques used in metamorphic generators; (ii) Instruction reordering (IR), which transposes instructions that do not depend on the output of previous instructions; (iii) Dead code insertion (DCI), which injects a piece of code that has no effect on program execution (i.e., may not

execute or may execute with no effect); and (iv) equivalent instruction replacement (EIR). We perform initial tests on the selected files and report the accuracy measurements. code transformation techniques are then applied and new accuracy is obtained and observed.

We used existing open-source tools for the C++ refactoring process [1, 16]. We consider the introduced techniques in 2.3. The results are shown in Table 6.5. The results shown in the table demonstrate that our system performs well in identifying similarities.

6.5.5 Time Efficiency

The running time for extracting the semantics of code is measured by considering the total time spent during each step: normalization process, extracting the semantics of the data flow, extracting the semantics of the control flow, and forming the SFG by extracting the binary relations. In the semantic extraction process, the binary application is first disassembled using IDA pro [8], and features are then extracted by running our IDApython script. The assembly instructions must first be normalized and hashed to a unique value. This process of extracting the features takes 15 seconds for the smallest application in our dataset (which is dumpcap) and 45 seconds for the largest application (libpng) on a Windows 32-bit machine with 16GB RAM. Extracting the first part of the semantics (data flow) takes 20 seconds for dumpcap and 60 seconds for libpng, while extracting the second part of the semantics (control flow) takes 23 seconds for dumpcap and 26 seconds for libpng. The last step, forming the new representation and extracting the relations described in Section 4.5, takes 10 seconds for dumpcap and 14 seconds for libpng.

Based on those results, we believe our system will be efficient enough for most real world applications.

Table 6.5: Results after applying code transformation techniques

Method	Precision	Recall	F1
RR	0.89	0.88	0.88
IR	0.91	0.92	0.91
DCI	0.87	0.93	0.90
EIR	0.81	0.82	0.81
RV	0.87	0.90	0.88
MM	0.85	0.82	0.83
NM	0.67	0.72	0.70

6.5.6 Applications

In this section, we demonstrate the applicability of our system to two applications: authorship attribution and clone detection. Previous work has demonstrated that it is possible to identify the authors of binary code [27, 112]. However, existing approaches usually assume that the compiler and its settings are known. In addition, the features used in such techniques are sensitive to any code transformation techniques. Hence, we apply our system to the binary and then re-examine their features based on the outputs of our system. Regarding clone detection, some existing works have demonstrated the use of K-CFG [81], Tracelet, n-grams [81, 112], idioms [81, 112], RFG [27], and strings [81]. Both authorship and clone accuracy are greatly improved by integrating our tool with the aforementioned tools, as shown in Table 6.6.

Dataset. The dataset we use for authorship attribution originates from Google Code Jam

2010 [5]. It consists of single-authored programs. For each author, there are multiple programs as the Code Jam is a multi-round programming contest. The dataset therefore provides a perfect benchmark for authorship attribution, and data from Google Code Jam has been used in all recent program authorship studies (e.g., [27], [112]). Regarding clone detection, we use 10 programs from our dataset (1-10).

Evaluation. Because the application domain is much more sensitive to false positives than false negatives, we use the F-measure as introduced in 5.4.5. Because each component in our system can handle one or more effects, our system could enhance the application of existing works. For instance, the normalization can handle compiler effects, data flow analysis can identify inline functions, the caller-callee relationship can tackle the refactoring process, and the relation extracted from the SFG can handle most code transformation techniques. Results are summarized in Table 6.6.

Table 6.6: Effect of integrating BinGold to certain existing works

Feature	$F_{0.5}$ **(Before applying BinGold)**	$F_{0.5}$ **(After applying BinGold)**	**Application**
Idioms [112]	0.71	0.80	Authorship
Idioms [81]	0.72	0.88	Clone
Graphlet [112]	0.60	0.76	Authorship
RFG [27]	0.72	0.79	Authorship
Call graphlet [112]	0.64	0.71	Authorship
K-CFG [81]	0.78	0.877	Clone
Tracelet [53]	0.66	0.70	Function Fingerprinting

According to the results in Table 6.6, we can conclude that our tool leads to substantial improvements in the accuracy of existing work. For instance, it improves the accuracy of clone systems (e.g., idioms) by 16%, which is a considerable improvement. Another

example considers the Tracelet system, since it already includes normalization techniques and data flow analysis, our tools only provide the benefit of semantics in terms of control flow graph, which leads to 4% improvement of accuracy.

6.6 Summary

To conclude, we have designed a system called *BinGold* for accurately and automatically recovering the semantics of a binary code. Our experimental results indicate that the approach is efficient in terms of computational resources and could thus be considered a practical approach to real-world binary analysis. Moreover, the experimental results suggest that *BinGold* can be used to enhance existing techniques, making them more robust and practical.

Chapter 7

Conclusion

This chapter concludes the findings of this thesis and highlights the future directions.

7.1 Concluding Remarks

The rise of malware attacks reported by companies and anti-virus vendors has pushed security researchers to propose new methodologies to extract intelligence about the authors of these attacks in order to provide countermeasures. In this context, this thesis aims to provide automated solutions for understanding the behavior of such malware binaries. We have elaborated on four threads of research, which may help provide interesting insights about malware binary code. We have shown how static and dynamic analyses of malware binary code help the security community to identify binary provenance, reused functions, third-party libraries such as free open source packages, and binary authorship attribution.

We began our research reviewing the existing binary code fingerprinting frameworks. Hence, we systematized the area of binary code fingerprints according to its most important dimensions: the applications that motivate its importance, the approaches used, and the aspects of the framework fingerprints. The details of this study was provided in Chapter 2. This step is important since it allowed us to investigate the different aspects of binary code to gain expertise in malware binary analysis and to define new perspectives related to malware research. Despite the importance of reverse-engineering prominent malware binary code analysis, this process turns to be tedious due to the huge number of observed malware collected in the wild. It also investigated the existing efforts that are related to binary program provenance, reused function detection, fingerprinting free open software packages, and binary authorship attribution.

In Chapter 3, we proposed a novel approach called *SIGMA* for effectively identifying reused functions in binary code. Instead of relying on one source of information, our approach combines multiple representations into one joint data structure *SIG*. *SIGMA* also supports inexact matching and exact matching based on traces of the *SIG* which deals with function fragments. Our experimental results demonstrated the effectiveness of our method.

In Chapter 4, we introduced FOSSIL, a system for identifying FOSS functions. It facilitates the tedious and error-prone task of manual malware reverse engineering and enables the use of suitable security tools on binary code. Determining FOSS functions in malware binaries has received limited attention compared to other fields such as clone

detection. Our evaluation demonstrated that *FOSSIL* yields highly accurate results.

In Chapter 5, we proposed *BinAuthor*, a system capable of decoupling program functionality from authors' coding habits in binary code. It leveraged a set of features that are based on collections of functionality-independent choices made by authors during coding. Our evaluation showed that *BinAuthor* outperforms existing methods in several aspects. First, decoupling authorship from functionality allows us to apply *BinAuthor* to real malware binaries to automatically generate evidence on similar coding habits, which matches existing findings by security experts and reverse engineers. Second, it successfully attributes a larger number of authors with significantly higher accuracy when compared to existing research contributions. Third, *BinAuthor* is more robust than previous methods in the sense that there is no significant drop in accuracy when the code is subjected to refactoring techniques, source and binary obfuscation, and different compilers.

In Chapter 6, we proposed a novel technique that extracts the semantics of binary code in terms of both data and control flow. Our technique allowed more robust binary analysis because the extracted semantics of the binary code are generally immune from code transformation, refactoring, and variations of the compilers or compilation settings. We applied data-flow analysis to extract the semantic flow of the registers as well as the semantic components of the control flow graph, which are then synthesized into a novel representation called the semantic flow graph (SFG). Our experimental results demonstrated that *BinGold* can be used to enhance existing techniques such as binary authorship attribution.

7.2 Future Directions

Our future work aims to include the following directions.

Advanced Obfuscation: This thesis is based on a main assumption that the binary code under analysis is unpacked and de-obfuscated. While this assumption may be reasonable for many general-purpose software, it implies the need for a pre-processing step involving unpacking/de-obfuscation before applying the method to malware. Second, our tool fails to handle most of the advanced obfuscation techniques such as Virtualization and jitting. We have plan to extend this thesis to include a set of dynamic features.

Privacy Concerns: Our tool, *BinAuthor*, could be misused to violate privacy of the coders. Therefore, we have to consider the privacy implications of *BinAuthor* in the future work.

Multiple Architecture: This thesis deals with only one architecture (x86). We chose this architecture because the most common CPU architectures nowadays are x86 for personal computers and server systems. However, in the world of mobile computing, the ARM architecture is the most common. The MIPS is also important in most control systems. We will study how to systematically address the problem of dealing with multiple architectures in future work.

Dataset Size: Although our current repository already has a decent size, it would need to be further enriched with a massive number of files. However, one of the biggest challenges we face involves how to automate gathering, compiling, and indexing FOSS packages. Each FOSS may have its unique dependencies, which makes automating the process

difficult. Our future research will include extending this system as a search engine for binary queries, and to also test it under a larger number of FOSS packages. Thus, a small fragment of assembly code or an executable could be queried to obtain useful information related to their functionality.

Efficiency: Through our experimental results, we notice that our tools, *SIGMA* and *BinGold*, the efficiency is an issue. Consequently, we have a plan in the future to employ the MapReduce paradigm [56] with a distributed version of the algorithm in order to distribute the computation over a cluster of servers.

Bibliography

[1] CodeRush for Visual Studio: Refactoring tool. https://www.devexpress.com/Products/CodeRush/. Last visited: Feb, 2018.

[2] Contagio: malware dump. http://contagiodump.blogspot.ca. Last visited: Feb, 2018.

[3] PE Packer detector for Windows . http://exeinfo.atwebpages.com/. Last visited: Feb, 2018.

[4] VirusSign: Malware Research & Data Center, Virus Free. http://www.virussign.com/. Last visited: Feb, 2017.

[5] Google Code Jam Contest Dataset. http://code.google.com/codejam/, 2008-2017. Last visited: Feb, 2018.

[6] Fast Library Identification and Recognition Technology. https://www.hex-rays.com/products/ida/tech/, 2011. Last visited: Mar, 2017.

[7] GitHub-Build software better. https://github.com/trending?l=cpp, 2011. Last visited: May, 2017.

[8] IDA Fast Library Identification and Recognition Technology. http://www.hex-rays.com/, 2011.

[9] Materials supplement for the paper "Who Wrote This Code? Identifying the Authors of Program Binaries". http://pages.cs.wisc.edu/~nater/esorics-supp/, 2011. Last visited: May, 2017.

[10] Mcafee: Technical report. www.mcafee.com/ca/resources/wp-citadel-trojan-summary.pdf, 2011. Last visited: Mar, 2017.

[11] Citizen Lab. https://citizenlab.org/, 2015. Last visited: Mar, 2017.

[12] Gephi plugin for nneo4j. Avaiable from:. https://marketplace.gephi.org/plugin/neo4j-graph-database-support/, 2015. Last visited: Feb, 2016.

[13] Materials supplement for the paper "Programmer De-anonymization from Binary Executables". https://github.com/calaylin/bda, 2015. Last visited: Jan, 2017.

[14] Planet source code. Available from:. http://www.planet-source-code.com/vb/default.asp?lngWId=3\ #ContentWinners, 2015. Last visited: Mar, 2017.

[15] Scalable Native Graph Database. Available from:. http://neo4j.com/, 2015. Last visited: Feb, 2016.

[16] C++ refactoring tools for visual studio. http://www.wholetomato.com/, 2016. Last visited: Feb, 2016.

[17] Gas Obfuscator. https://github.com/defuse/gas-obfuscation, 2016. Last visited: Mar, 2017.

[18] Gnu software repository. www.gnu.org/software/software.html, 2016. Last visited: Feb, 2016.

[19] HexRays: FLAIR, 2016. https//www.hex-rays.com/products/ida/support/download.shtml.

[20] Lintian Reports. lintian.debian.org, 2016. Last visited: Feb, 2018.

[21] Nynaeve: Adventure in Windows debugging and reverse enigineering. http://www.nynaeve.net/, 2016. Last visited: Mar, 2017.

[22] Oreans: Advanced Windows software protection system . http://www.oreans.com/themida.php, 2016. Last visited: Jan, 2017.

[23] PELock is a software security solution designed for protection of any 32 bit Windows applications . https://www.pelock.com/, 2016. Last visited: Jan, 2016.

[24] Sourceforge. http://sourceforge.net, 2016. Last visited: Feb, 2017.

[25] Tigress is a diversifying virtualizer/obfuscator for the C language. http://tigress.cs.arizona.edu/, 2016. Last visited: Jan, 2017.

[26] Tracelet system. https://github.com/Yanivmd/TRACY, 2016. Last visited: Feb, 2018.

[27] Saed Alrabaee, Noman Saleem, Stere Preda, Lingyu Wang, and Mourad Debbabi. Oba2: An onion approach to binary code authorship attribution. *Digital Investigation*, 11:S94–S103, 2014.

[28] Saed Alrabaee, Paria Shirani, Mourad Debbabi, and Lingyu Wang. On the feasibility of malware authorship attribution. In *International Symposium on Foundations and Practice of Security*, pages 256–272. Springer, 2016.

[29] Saed Alrabaee, Paria Shirani, Lingyu Wang, and Mourad Debbabi. Sigma: A semantic integrated graph matching approach for identifying reused functions in binary code. *Digital Investigation*, 12:S61–S71, 2015.

[30] Saed Alrabaee, Paria Shirani, Lingyu Wang, and Mourad Debbabi. Fossil: A resilient and efficient system for identifying foss functions in malware binaries. *ACM Transactions on Privacy and Security (TOPS)*, 21(2):8, 2018.

[31] Saed Alrabaee, Lingyu Wang, and Mourad Debbabi. Bingold: Towards robust binary analysis by extracting the semantics of binary code as semantic flow graphs (sfgs). *Digital Investigation*, 18:S11–S22, 2016.

[32] Dennis Andriesse, Asia Slowinska, and Herbert Bos. Compiler-agnostic function detection in binaries. In *Security and Privacy (EuroS&P), 2017 IEEE European Symposium on*, pages 177–189. IEEE, 2017.

[33] Dorian C Arnold, Dong H Ahn, Bronis R De Supinski, Gregory L Lee, Barton P Miller, and Martin Schulz. Stack trace analysis for large scale debugging. In *IEEE International Conference in Parallel and Distributed Processing Symposium (IPDPS)*, pages 1–10. IEEE, 2007.

[34] Thanassis Avgerinos, Sang Kil Cha, Brent Lim Tze Hao, and David Brumley. AEG: Automatic exploit generation. In *Network and Distributed System Security Symposium (NDSS)*, pages 283–300, 2011.

[35] Gogul Balakrishnan and Thomas Reps. Wysinwyx: What you see is not what you execute. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(6):23, 2010.

[36] Musard Balliu, Mads Dam, and Roberto Guanciale. Automating information flow analysis of low level code. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1080–1091. ACM, 2014.

[37] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, behavior-based malware clustering. In *Network and Distributed System Security Symposium (NDSS)*.

[38] B Bencsáth, G Pék, L Buttyán, and M Felegyhazi. skywiper (aka flame aka flamer): A complex malware for targeted attacks. *CrySyS Lab Technical Report, No. CTR-2012-05-31*, 2012.

[39] Boldizsár Bencsáth, Gábor Pék, Levente Buttyán, and Mark Felegyhazi. The cousins of stuxnet: Duqu, flame, and gauss. *Future Internet*, 4(4):971–1003, 2012.

[40] Daniel Bilar. Opcodes as predictor for malware. *International Journal of Electronic Security and Digital Forensics*, 1(2):156–168, 2007.

[41] Martial Bourquin, Andy King, and Edward Robbins. Binslayer: accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, page 4. ACM, 2013.

[42] Juan Caballero, Noah M Johnson, Stephen McCamant, and Dawn Song. Binary code extraction and interface identification for security applications. Technical report, DTIC Document. No. UCB/EECS-2009-133., 2009.

[43] Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. De-anonymizing programmers via code stylometry. In *USENIX Security 15*, pages 255–270, 2015.

[44] Aylin Caliskan-Islam, Fabian Yamaguchi, Edwin Dauber, Richard Harang, Konrad Rieck, Rachel Greenstadt, and Arvind Narayanan. When coding style survives compilation: De-anonymizing programmers from executable binaries. 2018.

[45] Shuang Cang and Derek Partridge. Feature ranking and best feature subset using mutual information. *Neural Computing & Applications*, 13(3):175–184, 2004.

[46] Silvio Cesare, Yang Xiang, and Wanlei Zhou. Control flow-based malware variant-detection. *IEEE Transactions on Dependable and Secure Computing*, 11(4):307–317, 2014.

[47] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. Bingo: cross-architecture cross-os binary search. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 678–689. ACM, 2016.

[48] Rudi Cilibrasi and Paul Vitanyi. Clustering by compression. *IEEE Transactions on Information Theory*, 51(4):1523–1545, 2005.

[49] Paolo Milani Comparetti, Guido Salvaneschi, Engin Kirda, Clemens Kolbitsch, Christopher Kruegel, and Stefano Zanero. Identifying dormant functionality in malware programs. In *IEEE Symposium on Security and Privacy (SP)*, pages 61–76. IEEE, 2010.

[50] Scott A Czepiel. Maximum likelihood estimation of logistic regression models: theory and implementation. 2002.

[51] Sanjeev Das, Yang Liu, Wei Zhang, and Mahintham Chandramohan. Semantics-based online malware detection: Towards efficient real-time protection against malware. *IEEE Transactions on Information Forensics and Security*, 11(2):289–302, 2016.

[52] Yaniv David, Nimrod Partush, and Eran Yahav. Statistical similarity of binaries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 266–280. ACM, 2016.

[53] Yaniv David and Eran Yahav. Tracelet-based code search in executables. In *ACM SIGPLAN Notices*, volume 49, pages 349–360. ACM, 2014.

[54] Jesse Davis and Mark Goadrich. The relationship between precision-recall and roc curves. In *Proceedings of the 23rd international conference on Machine learning*, pages 233–240. ACM, 2006.

[55] José Gaviria de la Puerta, Borja Sanz, Igor Santos, and Pablo García Bringas. Using dalvik opcodes for malware detection on android. In *Hybrid Artificial Intelligent Systems*, pages 416–426. Springer, 2015.

[56] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[57] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)*, 44(2):6, 2012.

[58] Manuel Egele, Maverick Woo, Peter Chapman, and David Brumley. Blanket execution: Dynamic similarity testing for program binaries and components. In *USENIX Security 14*, pages 303–317, 2014.

[59] Bruce S Elenbogen and Naeem Seliya. Detecting outsourced student programming assignments. *Journal of Computing Sciences in Colleges*, 23(3):50–57, 2008.

[60] Ammar Ahmed E Elhadi, Mohd Aizaini Maarof, Bazara IA Barry, and Hentabli Hamza. Enhancing the detection of metamorphic malware using call graphs. *Computers & Security*, 46:62–78, 2014.

[61] Khaled ElWazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. Scalable variable and data type detection in a binary rewriter. In *ACM SIGPLAN Notices*, volume 48, pages 51–60. ACM, 2013.

[62] Wenbin Fang, Barton P Miller, and James A Kupsch. Automated tracing and visualization of software security structure and properties. In *Proceedings of the ninth international symposium on visualization for cyber security*, pages 9–16. ACM, 2012.

[63] Mohammad Reza Farhadi, Benjamin CM Fung, Philippe Charland, and Mourad Debbabi. Binclone: Detecting code clones in malware. In *Eighth International Conference on Software Security and Reliability*, pages 78–87. IEEE, 2014.

[64] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of*

the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS), pages 480–491. ACM, 2016.

[65] Eric Filiol and Sébastien Josse. A statistical model for undecidable viral detection. *Journal in Computer Virology*, 3(2):65–74, 2007.

[66] Halvar Flake. Graph-based binary analysis. *Blackhat Briefings 2002*, 2002.

[67] Martin Fowler. *Refactoring: improving the design of existing code*. Pearson Education India, 1999.

[68] Carlos Gañán, Orcun Cetin, and Michel van Eeten. An empirical analysis of zeus c&c lifetime. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, pages 97–108. ACM, 2015.

[69] Thomas Gärtner, Peter Flach, and Stefan Wrobel. On graph kernels: Hardness results and efficient alternatives. In *Learning Theory and Kernel Machines*, pages 129–143. Springer, 2003.

[70] William H Hawkins, Jason D Hiser, Michele Co, Anh Nguyen-Tuong, and Jack W Davidson. Zipr: Efficient static binary rewriting for security. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 559–566. IEEE, 2017.

[71] Xin Hu, Tzi-cker Chiueh, and Kang G Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM conference on Computer and communications security (CCS)*, pages 611–620. ACM, 2009.

[72] He Huang, Amr M Youssef, and Mourad Debbabi. Binsequence: Fast, accurate and scalable binary code reuse detection. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security (AsiaCCS)*, pages 155–166. ACM, 2017.

[73] Emily R Jacobson, Andrew R Bernat, William R Williams, and Barton P Miller. Detecting code reuse attacks with a model of conformant program execution. In *Engineering Secure Software and Systems*, pages 1–18. Springer, 2014.

[74] Emily R Jacobson, Nathan Rosenblum, and Barton P Miller. Labeling library functions in stripped binaries. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, pages 1–8. ACM, 2011.

[75] Jiyong Jang, David Brumley, and Shobha Venkataraman. Bitshred: feature hashing malware for scalable triage and semantic analysis. In *Proceedings of the 18th ACM conference on Computer and communications security (CCS)*, pages 309–320. ACM, 2011.

[76] Yoon-Chan Jhi, Xinran Wang, Xiaoqi Jia, Sencun Zhu, Peng Liu, and Dinghao Wu. Value-based program characterization and its application to software plagiarism detection. In *Proceedings of the 33rd International Conference on Software*

Engineering, pages 756–765. ACM, 2011.

[77] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM – software protection for the masses. In Brecht Wyseur, editor, *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, pages 3–9. IEEE, 2015.

[78] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-llvm: software protection for the masses. In *Proceedings of the 1st International Workshop on Software Protection*, pages 3–9. IEEE Press, 2015.

[79] Tommi A Junttila and Petteri Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In *Proceedings of the Ninth Workshop on Algorithm Engineering and Experiments (ALENEX)*, volume 7, pages 135–149. SIAM, 2007.

[80] Anatoli Kalysch, Johannes Götzfried, and Tilo Müller. Vmattack: Deobfuscating virtualization-based packed binaries. page 2, 2017.

[81] Wei Ming Khoo, Alan Mycroft, and Ross Anderson. Rendezvous: a search engine for binary code. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 329–338. IEEE Press, 2013.

[82] Donald E Knuth. Backus normal form vs. backus naur form. *Communications of the ACM*, 7(12):735–736, 1964.

[83] Ivan Krsul and Eugene H Spafford. Authorship analysis: Identifying the author of a program. *Computers & Security*, 16(3):233–257, 1997.

[84] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Polymorphic worm detection using structural information of executables. In *Recent Advances in Intrusion Detection (RAID)*, pages 207–226. Springer, 2005.

[85] Kaspersky Lab. Resource 207: Kaspersky Lab Research proves that Stuxnet and Flame developers are connected. `http://newsroom.kaspersky.eu/fileadmin/user_upload/en/Images/Lifestyle/20120611_Kaspersky_Lab_Press_Release_Flame_Stuxnet_cooperation_final_-_UK.pdf`, 2012. Last visited: Feb, 2018.

[86] Arun Lakhotia, Mila Dalla Preda, and Roberto Giacobazzi. Fast location of similar code fragments using semantic'juice'. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop*, page 5. ACM, 2013.

[87] Da Lin and Mark Stamp. Hunting for undetectable metamorphic viruses. *Journal in computer virology*, 7(3):201–214, 2011.

[88] Martina Lindorfer, Alessandro Di Federico, Federico Maggi, Paolo Milani Comparetti, and Stefano Zanero. Lines of malicious code: insights into the malicious software industry. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC)*, pages 349–358. ACM, 2012.

[89] Fan Long, Stelios Sidiroglou-Douskos, and Martin Rinard. Automatic runtime error repair and containment via recovery shepherding. In *ACM SIGPLAN Notices*, volume 49, pages 227–238. ACM, 2014.

[90] Prasanta Chandra Mahalanobis. On the generalized distance in statistics. *Proceedings of the National Institute of Sciences (Calcutta)*, 2:49–55, 1936.

[91] Marion Marschalek. Big Game Hunting: Nation-state malware research, BlackHat. `https://www.blackhat.com/docs/webcast/08202015-big-game-hunting.pdf/`, 2015. Last visited: Feb, 2018.

[92] Ryan McDonald and Fernando Pereira. Identifying gene and protein mentions in text using conditional random fields. *BMC bioinformatics*, 6(1):1, 2005.

[93] Xiaozhu Meng and Barton P Miller. Binary code is not easy. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 24–35. ACM, 2016.

[94] Xiaozhu Meng, Barton P Miller, and Kwang-Sung Jun. Identifying multiple authors in a binary program. In *European Symposium on Research in Computer Security (ESORICS)*, pages 286–304. Springer, 2017.

[95] Barton P Miller, Mark D Callaghan, Jonathan M Cargille, Jeffrey K Hollingsworth, R Bruce Irvin, Karen L Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tool. *Computer*, 28(11):37–46, 1995.

[96] Jiang Ming, Meng Pan, and Debin Gao. ibinhunt: Binary hunting with inter-procedural control flow. In *Information Security and Cryptology (ICISC)*, pages 92–109. Springer, 2013.

[97] Ned Moran and James Bennett. *Supply Chain Analysis: From Quartermaster to Sun-shop*, volume 11. FireEye Labs, 2013.

[98] James Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics*, 5(1):32–38, 1957.

[99] Emerson Murphy-Hill, Chris Parnin, and Andrew P Black. How we refactor, and how we know it. *IEEE Transactions on Software Engineering*, 38(1):5–18, 2012.

[100] Robert Muth. Register liveness analysis of executable code. *Manuscript, Dept. of Computer Science, The University of Arizona, Dec*, 1998.

[101] Lakshmanan Nataraj, Dhilung Kirat, BS Manjunath, and Giovanni Vigna. Sarvam: Search and retrieval of malware. In *Proceedings of the Annual Computer Security Conference (ACSAC) Worshop on Next Generation Malware Attacks and Defense (NGMAD)*, 2013.

[102] Lina Nouh, Ashkan Rahimian, Djedjiga Mouheb, Mourad Debbabi, and Aiman Hanna. Binsign: Fingerprinting binary functions to support automated analysis of code executables. In *IFIP International Conference on ICT Systems Security and Privacy Protection*, pages 341–355. Springer, 2017.

[103] Pádraig OáSullivan, Kapil Anand, Aparna Kotha, Matthew Smithson, Rajeev Barua, and Angelos D Keromytis. Retrofitting security in cots software with binary rewriting. In *Future Challenges in Security and Privacy for Academia and Industry*, pages 154–172. Springer, 2011.

[104] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *IEEE Symposium on Security and Privacy (SP)*, pages 601–615. IEEE, 2012.

[105] Hanchuan Peng, Fuhui Long, and Chris Ding. Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(8):1226–1238, 2005.

[106] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. Cross-architecture bug search in binary executables. In *IEEE Symposium on Security and Privacy*, pages 709–724. IEEE, 2015.

[107] Qiu, Xiaohong Su, and Peijun Ma. Using reduced execution flow graph to identify library functions in binary code. *IEEE Transactions on Software Engineering*, 42(2):187–202, 2016.

[108] Ashkan Rahimian, Paria Shirani, Saed Alrbaee, Lingyu Wang, and Mourad Debbabi. Bincomp: A stratified approach to compiler provenance attribution. *Digital Investigation*, 14:S146–S155, 2015.

[109] Václav Rajlich. Software evolution and maintenance. In *Proceedings of the Future of Software Engineering*, pages 133–144. ACM, 2014.

[110] Alexandre Rebert, Sang Kil Cha, Thanassis Avgerinos, Jonathan Foote, David Warren, Gustavo Grieco, and David Brumley. Optimizing seed selection for fuzzing. In *USENIX Security*, pages 861–875, 2014.

[111] Kaspar Riesen, Xiaoyi Jiang, and Horst Bunke. Exact and inexact graph matching: Methodology and applications. *Managing and Mining Graph Data*, pages 217–247, 2010.

[112] Nathan Rosenblum, Xiaojin Zhu, and Barton P Miller. Who wrote this code? identifying the authors of program binaries. In *European Symposium on Research in Computer Security (ESORICS)*, pages 172–189. Springer, 2011.

[113] Nathan E Rosenblum. *The Provenance Hierarchy of Computer Programs*. PhD thesis, University of Wisconsin–Madison, 2011.

[114] Nathan E Rosenblum, Barton P Miller, and Xiaojin Zhu. Extracting compiler provenance from program binaries. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 21–28. ACM, 2010.

[115] Brian Ruttenberg, Craig Miles, Lee Kellogg, Vivek Notani, Michael Howard, Charles LeDoux, Arun Lakhotia, and Avi Pfeffer. Identifying shared software components to support malware forensics. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 21–40. Springer, 2014.

[116] Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. Recognizing functions in binaries with neural networks. In *USENIX Security*, pages 611–626, 2015.

[117] Paria Shirani, Lingyu Wang, and Mourad Debbabi. Binshape: Scalable and robust binary library function identification using function shape. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 301–324. Springer, 2017.

[118] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *IEEE Symposium on Security and Privacy (SP)*, pages 138–157. IEEE, 2016.

[119] Eugene H Spafford and Stephen A Weeber. Software forensics: Can we track code to its authors? *Computers & Security*, 12(6):585–595, 1993.

[120] Mark Stamp. A revealing introduction to hidden markov models. *Department of Computer Science San Jose State University*, 2004.

[121] Annie H Toderici and Mark Stamp. Chi-squared distance and metamorphic virus detection. *Journal of Computer Virology and Hacking Techniques*, 9(1):1–14, 2013.

[122] S Vichy N Vishwanathan, Nicol N Schraudolph, Risi Kondor, and Karsten M Borgwardt. Graph kernels. *The Journal of Machine Learning Research*, 11:1201–1242, 2010.

[123] Jason Tsong-Li Wang, Qicheng Ma, Dennis Shasha, and Cathy H. Wu. New techniques for extracting features from protein sequences. *IBM Systems Journal*, 40(2):426–441, 2001.

[124] Zheng Wang, Ken Pierce, and Scott McFarling. Bmat-a binary matching tool for stale profile propagation. *The Journal of Instruction-Level Parallelism*, 2:1–20, 2000.

[125] Zhenchang Xing and Eleni Stroulia. Refactoring practice: How it is and how it should be supported-an eclipse case study. In *22nd IEEE International Conference on Software Maintenance (ICSM'06)*, pages 458–468. IEEE, 2006.

[126] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *IEEE Symposium on Security and Privacy (SP)*, pages 590–604. IEEE, 2014.

[127] Chaitanya Yavvari, Arnur Tokhtabayev, Huzefa Rangwala, and Angelos Stavrou. Malware characterization using behavioral components. In *International Conference on Mathematical Methods, Models, and Architectures for Computer Network Security*, pages 226–239. Springer, 2012.

[128] Yanfang Ye, Tao Li, Donald Adjeroh, and S Sitharama Iyengar. A survey on malware detection using data mining techniques. *ACM Computing Surveys (CSUR)*, 50(3):41, 2017.

[129] Yanfang Ye, Tao Li, Yong Chen, and Qingshan Jiang. Automatic malware categorization using cluster ensemble. In *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 95–104. ACM, 2010.

[130] Ilsun You and Kangbin Yim. Malware obfuscation techniques: A brief survey. In *International Conference on Broadband, Wireless Computing, Communication and Applications (BWCCA)*, pages 297–300. IEEE, 2010.

[131] Li Yujian and Liu Bo. A normalized levenshtein distance metric. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(6):1091–1095, 2007.

[132] Yijia Zhang, Hongfei Lin, Zhihao Yang, and Yanpeng Li. Neighborhood hash graph kernel for protein–protein interaction extraction. *Journal of biomedical informatics*, 44(6):1086–1092, 2011.

[133] Yijia Zhang, Hongfei Lin, Zhihao Yang, Jian Wang, and Yanpeng Li. Hash subgraph pairwise kernel for protein-protein interaction extraction. *IEEE/ACM Transactions on Computational Biology and Bioinformatics (TCBB)*, 9(4):1190–1202, 2012.