BINSIGN: FINGERPRINTING BINARY FUNCTIONS TO SUPPORT AUTOMATED ANALYSIS OF CODE EXECUTABLES

Lina Nouh

A thesis
in
The Department
of
The Concordia Institute for Information Systems Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Applied Science in Information Systems Security
Concordia University
Montréal, Québec, Canada

January 2017

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Lina Nouh**

Entitled: **BinSign: Fingerprinting Binary Functions to Support Automated Analysis of Code Executables**

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

Master of Applied Science in Information Systems Security

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

Signed by the final examining commitee:

Dr. Chadi Assi _____ Chair

Dr. Amr Youssef _____ Examiner

Dr. Olga Ormandjieva _____ Examiner

Dr. Mourad Debbabi _____ Supervisor

Dr. Aiman Hanna _____ Supervisor

Approved _____

Chair of Department or Graduate Program Director

_____ 20 _____ _____

Dr. Amir Asif, Dean

Faculty of Engineering and Computer Science

Abstract

BinSign: Fingerprinting Binary Functions to Support Automated Analysis of Code Executables

Lina Nouh

Software reverse engineering is a complex process that incorporates different techniques involving static and dynamic analyses of software programs. Numerous tools are available that help reverse engineers in automating the dynamic analysis process. However, the process of static analysis remains a challenging and tedious process for reverse engineers. The static analysis process requires a great amount of manual work. Therefore, it is very demanding and time-consuming. One aspect of reverse engineering that provides reverse engineers with useful information regarding a statically analyzed piece of code is function fingerprinting. Binary code fingerprinting is a challenging problem that requires an in-depth analysis of internal binary code components for deriving identifiable and expressive signatures.

Binary code fingerprints are helpful in the reverse engineering process and have various security applications such as malware variant detection, malware clustering, binary auditing, function recognition, and library identification. Moreover, binary code fingerprinting is also useful in automating some reverse engineering tasks such as clone detection, library function identification, code similarity, authorship attribution, etc. In addition, code fingerprints are valuable in cyber forensics as well as the process of patch analysis in order to identify patches or make sure that the patch complies with the security requirements.

In this thesis, we propose a binary function fingerprinting and matching approach and implement a tool named BinSign based on the proposed approach that enhances and accelerates the reverse engineering process. The main objective of BinSign is to provide an accurate and scalable solution to binary code fingerprinting by computing and matching structural and syntactic code profiles for disassemblies while outperforming existing techniques. The structural profile of binary code is captured through decomposing the control-flow-graph of a function into tracelets. We describe the underlying methodology and evaluate its performance in several use cases, including function matching, function reuse, library function detection, malware analysis, and

function indexing scalability. We also provide some insights into the effects of different optimization levels and obfuscation techniques on our fingerprint matching methodology. Additionally, we emphasize the scalability aspect of BinSign that is achieved through applying locality sensitive hashing, filtering techniques, and distributing the computations across several machines. The min-hashing process is combined with the banding technique of locality sensitive hashing in order to ensure a scalable and efficient fingerprint matching process. We perform our experiments on a database of 6 million functions that includes well-known libraries, malware samples, and some dynamic library files obtained from the Microsoft Windows operating system. The indexing process of fingerprints is distributed across multiple machines and it requires an average time of 0.0072 seconds per function. A comparison is also conducted with relevant existing tools, which shows that BinSign achieves a higher accuracy than these tools.

Acknowledgments

Foremost, I thank God for granting me the will power and dedication necessary to complete my studies and compose this thesis.

I would like to express my sincere appreciation to everyone who provided me with help and support during the course of my research.

I owe my deepest gratitude to my supervisors, Dr. Mourad Debbabi and Dr. Aiman Hanna, for their wise guidance and continuous encouragement throughout my research journey. Their constant support, enlightened supervision, and knowledgeable advice had a great significance in making this thesis possible.

My gratitude extends to the examining committee members: Dr. Amr Youssef and Dr. Olga Ormandjieva. I highly appreciate the time and effort spent to evaluate my thesis. It is such a privilege to have them as the members of my examining committee.

I would also like to extend my gratitude to all my colleagues and lab-mates at Concordia University for their generous help in overcoming any challenges that faced me during my research. I owe special thanks to Djedjiga Mouheb, a great friend who also worked with me closely and provided valuable assistance and collaboration whenever needed throughout my work.

Last but not least, I owe special gratitude and appreciation to my family and friends who provide me with endless encouragement and moral support. I would like to particularly thank my parents, who are the reason of my existence and to whom I owe any accomplishments in my life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivations

Reverse engineering is a common approach to recover design-level abstractions from an unknown (target) code or system and understand, as much as possible, its functionality, architecture, and the inner workings of its internal components [75]. Most of the time, the target system is represented by a binary program, consisting of a set of modules, instructions, basic blocks, data and code sequences. The reverse engineering process entails a series of interleaving steps for static and dynamic analyses [72]. During the initial phases of this process, efforts are made for gaining essential information about the potential capabilities and objectives of the binary sample. One benefit of this information is in pinpointing the target of suspicious code and directing the detection and mitigation processes of infected systems [69].

One of the critical topics in computer security is malware analysis [20,38,62,73,79]. It has gained unprecedented attention due to an ever increasing array of cyber threats.

Over the past few years, the number and complexity of malware attacks have grown significantly. In 2015, around 431 million new malware variants were uncovered [6]. In order to address and remediate the emerging threats, advanced techniques and tools are required to support automated reverse engineering and malware analysis. One example of such techniques is binary fingerprinting.

Program disassembly [18] is an important phase in reverse engineering. This phase results in the binary code being organized into a set of assembly functions. Often, an in-depth and time-consuming process of analyzing the assembly code is conducted before detailed conclusions can be drawn on the code's functionality. In addition, analyzing assembly code entails a great deal of manually intensive work and requires supervised intervention, which hinders the overall analysis process. For example, today's proliferation of malware combined with its increasing sophistication have complicated the reverse engineering process and made malware analysis even more challenging. Moreover, it is quite common for new malware variants to improve on the previous ones or reuse their code fragments. Analyzing such malware requires correlating different malware samples to identify their similarities and the code fragments they share. This would, for instance, prevent reanalyzing malware code, which has already been analyzed.

Furthermore, binary function fingerprinting has numerous uses such as compiler identification [68, 70], library function identification, authorship analysis [13, 14, 74], clone detection, vulnerability detection, provenance analysis, malware detection and classification, etc. Additionally, binary code fingerprints are beneficial in cyber forensics [43] and patch analysis with the purpose of identifying the patch or ensuring the

compliance of the patch to security requirements. Although clone detection is one application of function fingerprinting, function fingerprinting can be used for several other purposes. Another advantage of binary fingerprinting is the ability to share information about a binary function by sharing the fingerprint without the need to share the binary piece of code itself since a binary function's fingerprint conveys valuable information that describes different aspects of the function.

1.2 Objectives

The main objective of this thesis consists of defining an approach for binary function fingerprinting and matching. This is achieved by designing an approach that integrates important features of binary functions to produce a meaningful function fingerprint. Additionally, the approach should allow for matching a target fingerprint against a large repository of fingerprints and calculating a similarity score between two fingerprints. In particular, this thesis aims at:

- Conducting a comparative study of the state-of-the-art techniques in function fingerprinting and matching.
- Elaborating a framework for the generation and matching of binary function fingerprints.
- Designing and implementing the proposed framework.
- Validating and evaluating the proposed approach through different use cases and experiments.

1.3 Contributions

The main contributions of this thesis are as follows:

- We propose a fingerprinting approach for binary functions using features that capture syntactic, semantic, and structural information of a function. We capture the structural information of a function's CFG through features of partial execution traces, i.e., tracelets.

- We design and implement an efficient and scalable matching framework to match a target function fingerprint against a large repository of fingerprints. The matching framework is capable of performing exact and inexact fingerprint matching, which makes it resilient to alterations in the binary functions presented by different compilers or optimization levels.

- We construct a multi-layer filtering mechanism in order to facilitate and expedite the fingerprint matching process.

- We design and implement a distribution mechanism for BinSign to improve scalability and performance.

- We evaluate BinSign in several use cases to illustrate the efficiency and effectiveness of our methodology, including function reuse, malware analysis, resilience to different optimization levels, resilience to obfuscation, and function indexing scalability.

1.4 Thesis Organization

The remainder of the thesis is structured as follows. Chapter 2 presents a literature review and background knowledge of different approaches to binary function fingerprinting. Chapter 3 provides details regarding the inner workings and the underlying algorithmics of the main components of our binary fingerprinting and matching methodologies, including the details of the fingerprint generation and matching algorithms. Experiments conducted in order to evaluate our methodologies are conveyed and discussed in Chapter 4. In Chapter 5, we ultimately present concluding remarks and a discussion of potential future research directions.

Chapter 2

Background and Related Work

This chapter presents an overview of function fingerprinting by providing background knowledge and elaborating on the different aspects of the process. The aim is to achieve a thorough understanding of the process of function fingerprinting, its multiple application domains, and the principal problems that need to be addressed in order to successfully compose an effective function fingerprinting solution. This chapter also provides an analysis of the most relevant research works. Function fingerprinting overlaps with different research areas such as clone detection since the fingerprints can be used to detect function clones by means of detecting similar fingerprints. Another overlapping research area is search-based fingerprinting as the fingerprint matching process mimics a search-based problem. Therefore, the reviewed related work in this chapter includes exact and inexact approaches to signature matching, graph-based analysis, clone detection, and search-based fingerprinting. In order to understand the process of function fingerprinting and the associated domains, we introduce a taxonomy that makes it more comprehensible.

2.1 Background Definitions

In order to properly understand the process of function fingerprinting, this section provides some definitions that are relevant to this domain. This section also presents some useful descriptions of features that can be used to characterize programs as provided in [26, 44].

- **Binary Instructions**: A binary instruction represents the basic computation unit. These computations can incorporate various operations such as unary and binary operations, procedure or library calls. One binary instruction can be defined by its mnemonic (opcode) and operand(s).

- **Leaders**: The set of leaders of a program comprise of: (a) The first statement at the beginning of the program, (b) The target statement of any goto statement, and (c) The first statement that immediately follows any goto statement.

- **Basic Blocks**: A basic block is a node in a function's CFG that starts with a leader and encapsulates a sequence of instructions such that: (a) Execution flow can reach the basic block only by going through its first instruction and (b) Execution flow can leave the basic block only at its last instruction.

- **Control-Flow-Graph**: The Control-Flow-Graph (CFG) of a function is a directed graph that illustrates the potential execution flow within a function. The nodes in the CFG symbolize basic blocks that contain a collection of instructions, which always start with a leader. The edges in the CFG represent the control flow between the basic blocks.

- **API Calls**: An API call depicts a call within an instruction to libraries or other imports.
- **Syntactic Features**: The features that result from analyzing the raw code of a program directly. In the case of analyzing source code, the raw code is represented by the textual stream. It might be normalized by eliminating comments and whitespace. In the case of analyzing binaries, the raw code is represented by the byte sequences of the binary instructions.
- **Semantic Features**: The features that describe the meaning or derived from a semantic representation of the program at hand.
- **Code Obfuscation**: Code obfuscation is the process of changing a piece of code in a manner that makes the code more difficult to understand while preserving its functionality. Obfuscation techniques are used in order to obstruct the analysis and reverse engineering of a software program.
- **Function Fingerprinting**: The process of function fingerprinting constructs a unique and compact representation of the functionality of a function. Functions that carry out equivalent functionalities should be assigned similar fingerprints. Moreover, the resulting fingerprints should be robust to byte-level discrepancies and tolerant to minor variations in the function. Furthermore, the likelihood of fingerprint collisions of different functions must be minimal.
- **Fingerprint Collision**: Fingerprint collision refers to the occurrence of identical fingerprints resulting from the fingerprint generation of two different functions.

2.2 Function Fingerprinting Taxonomy

In order to better understand function fingerprinting, we introduce a classification using a taxonomy that simplifies the understanding of the associated domains. Further on, we elaborate a taxonomy of the main issues raised by function fingerprinting and the solutions designed to address them. Our proposed taxonomy displayed in Figure 1 classifies current function fingerprinting research into three major areas, namely, fingerprint generation, fingerprint matching, and the different applications of fingerprinting. In the following, we elaborate on the proposed taxonomy. It is worth noting that different research works might refer to fingerprints using various equivalent terms such as signatures or simply a collection of features.

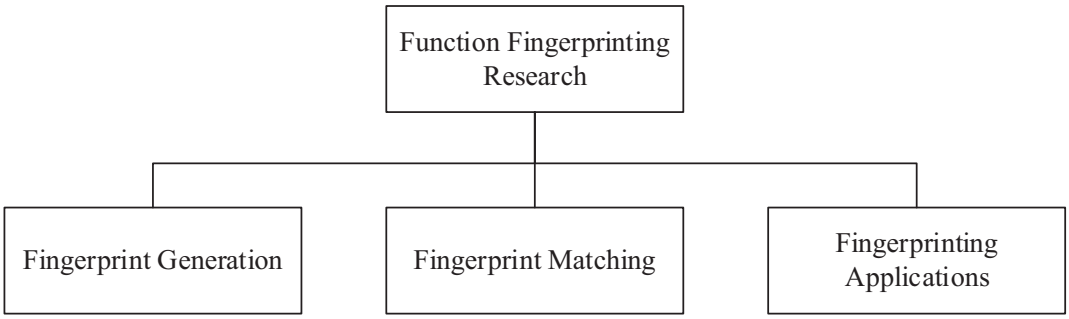


Figure 1: Function Fingerprinting Research Taxonomy

2.3 Fingerprint Generation

Function fingerprint generation involves extracting features of various types from the function at hand. Different research works use different types of features during the fingerprint generation process. The extracted features include semantic and syntactic

features, which might be hashed in order to produce a more compact representation. Some research works also focus on graph-based features of the function. Figure 2 summarizes our proposed fingerprint generation taxonomy.

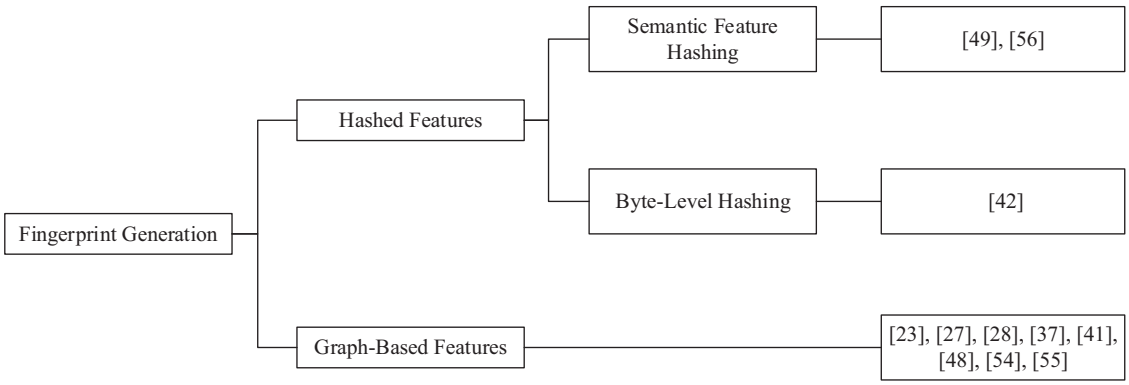


Figure 2: Fingerprint Generation Taxonomy

2.3.1 Hashed Features

Different studies use various techniques to generate fingerprints that meaningfully represent pieces of code. Part of these works use hashing methods on extracted features to produce function fingerprints that are compact and easily comparable [42, 49, 56].

Semantic Feature Hashing

The work in [56] uses the term "juice" to refer to an abstraction of semantics of a piece of binary code. A tool is implemented for extracting and hashing the juice of a binary program. The tool extracts the semantics of basic blocks through the process of symbolic interpretation. This is performed by extracting the effects of the basic block on

the state of the program. The process of extracting this semantic juice entails replacing register names and literal constants by typed, logical variables while maintaining algebraic constraints between the numeric variables. The resulting semantic template presumably remains identical despite of possible small code variations such as register renaming, memory address allocation, and constant replacement. Therefore, similar binary code fragments can be matched by conducting structural comparisons of the extracted juice or by comparing their hash values. However, this approach also has some limitations. The most notable limitation is that the main focus of this method is placed on the semantics of individual basic blocks. Different compilers and different optimization levels for example may cause variations in the contents and structure of the basic blocks in a function's CFG.

Another research work that uses semantic hashes to generate a fingerprint that captures the semantic information of a function is presented in [49]. The set of semantic features is used to cluster functions through the means of locality sensitive hashing to group functions with similar features together. The approach relies on the min-hashing technique to calculate an estimate of the distance between sets. Therefore, each function is reduced to a set of features. Each function is first disassembled, then the CFG is constructed. The features describing each function represent the input-output behavior of the basic blocks in that function's CFG. The function's execution state is considered to be its effects on values assigned to a set of registers and memory. Therefore, the considered input-output behavior of a basic block can be divided into four constituents: (i) the effects on registers; (ii) the effects on memory; (iii) the arguments being passed through a function call at the end of the basic block;

(iv) the condition of any jump instruction at the end of the basic block. These represent the semantic features that are hashed to generate a fingerprint that represents each function.

Byte-Level Hashing

Other tools use byte-level hashing techniques for fingerprint generation such as IDA F.L.A.I.R [42], which is a technology provided by *IDA Pro* [4]. IDA Pro is an industry-standard disassembler that is widely used by reverse engineering practitioners [48,55]. IDA Pro provides a mechanism known as IDA F.L.A.I.R (Fast Library Acquisition for Identification and Recognition) for generating new signatures for binary functions from non-standard libraries. The necessary data for applying the recognition algorithm is retained in a signature file. Every function is denoted by a different pattern. Each pattern consists of the foremost 32 bytes of a function such that any variant bytes are marked. Table 1 shows examples of function patterns as provided in [42].

Pattern	**Function Name**
558BEC0EFF7604.....59595DC3558BEC0EFF7604.....59595DC3	_registerbgidriver
558BEC1E078A66048A460E8B5E108B4E0AD1E9D1E980E1C0024E0C8A6E0A8A76	_biosdisk
558BEC1EB41AC55604CD211F5DC3.....	_setdta
558BEC1EB42FCD210653B41A8B5606CD21B44E8B4E088B5604CD219C5993B41A	_findfirst

Table 1: Examples of Function Patterns in IDA F.L.I.R.T.

In the previous pattern examples, the characters ".." represent variant bytes. As presented in these examples, some functions begin with the same sequence of bytes. As a result, a tree structure is used to store these functions efficiently. For instance, the tree structure in Figure 3 would be used to store the patterns of the functions in the previous examples.

Figure 3: Example of Patterns Tree Structure in IDA F.L.I.R.T.

The tree nodes hold partial byte sequences. In this instance, the root of the tree holds the first byte sequence "558BEC". Branching from the root node are three subtrees starting with the bytes 0E, 1E, B4 respectively. Two other subtrees then branch from the subtree starting with B4. At the ends of every subtree lie the leaf nodes. These leaves carry the information about the function. In this example, only the name of the function is shown.

The tree data structure is utilized in order to fulfill the following two objectives:

- The necessary memory space is reduced as bytes that are common to a number of functions are saved in tree nodes. The decrease in required memory space is relative to the number of functions that begin with the same byte sequence.

- The tree structure is fitting for the purpose of fast and efficient pattern matching. The number of necessary comparisons that need to be carried out in order to

match a particular location in a program to all the functions in a signature file increases logarithmically as the number of functions in the file increase.

2.3.2 Graph-Based Features

A number of research works have been carried out that rely on graph-based features for function fingerprint generation (e.g., [23, 27, 28, 41, 48, 54, 55]). These works use graph features for various purposes including similarity measurement, patch analysis, and malware detection. These works are inspired by a very well-known graph-based approach to binary comparison, which is proposed in [37]. This approach represents binary functions as directed graphs (generated from the call graph (CG) or CFG of a function), and focuses on structural features instead of the instruction sequence in the code. This results in tolerating minor byte-level differences such as instruction reordering. Furthermore, the graph-based approach is able to capture the flow and structure related features of binary programs more effectively. This approach abstracts each basic block as a tuple that contains features such as the number of graph nodes, edges, and function calls.

One limitation of this approach is placing all the attention on structural information, ignoring the instruction semantics of each basic block. Two functions that perform the same types of operations using a different control structure (perhaps due to instruction reordering) may be represented with very different fingerprints by this approach. An enhanced fingerprinting approach should take into account the types of instructions alongside structural encodings as part of each basic block tuple.

The work in [27] proposes a malware fingerprinting process that is defined by the

set of CFGs contained in a piece of malware code. Fingerprints are generated based on feature vectors of string-based signatures. The feature vectors are produced either by decomposing the set of graphs into k-subgraphs of a fixed size, or q-gram strings of the high-level source after decompilation. Several experiments are performed to evaluate different methods of generating the signatures. First, an experimental method of generating signatures through constructing strings is evaluated. Then, this method is discarded in favor of vector-based signatures, which are found to be more effective and efficient than string-based signatures.

In order to generate the signatures, the malware code is first unpacked to undo any obfuscation effects. The disassembly is transformed into an intermediate language representation. The control flow is reconstructed into a CFG for each procedure based on the intermediate code [25]. The CFGs are then normalized to remove unneeded jumps such as unconditional branching instructions that are used for splitting a basic block into two partial, equivalent basic blocks. This is an example of one obfuscation technique that can be used in a malware in order to change its byte-level content and static string signature. Then, the CFG of each procedure is decompiled and transformed into a string representation.

One of two algorithm options is used to extract the CFG features. The first possibility is to extract subgraphs of size k to represent the features. Alternatively, the strings representing the structured graphs are decomposed into q-grams that represent the extracted features. When using strings from decompiled CFGs, q-grams are analogous to n-grams. Either algorithm is used for feature extraction, the feature vector that represents the fingerprint is then constructed using the most

relevant features. In order to detect candidates that highly resemble existing malware samples, Vantage Point trees [78] are used to perform a metric similarity search.

Some research works use call-graph-based features for malware detection and clustering such as [41] and [55]. A method for Android malware detection based on efficient embeddings of function call graphs is presented in [41]. This research work explores how the process of machine learning classification of graphs can be efficiently applied to solve the problem of malware detection. The proposed methodology involves efficiently embedding call graphs with an explicit feature map inspired by a linear-time graph kernel. The proposed fingerprinting methodology is purely based on structural features of the call graph. The resulting method can be summarized in four main steps: (1) Call graph extraction and labeling, (2) Hashing of each node in the function call graph and direct neighboring nodes, (3) Feature space embedding using an explicit map inspired by [45], and (4) Machine learning and feature analysis processes are used to build a detection model that is able to classify different applications as either benign or malicious based on the previously extracted graph features.

Another malware classification proposal based on call graph clustering is presented in [55]. This approach relies on abstracting certain variations in malware samples by representing them as call graphs. The call graph representation enables the detection of structural similarities between malware samples. The aim is to apply more generic detection techniques through clustering similar malware samples together, thereby focusing on the common features of the samples within a certain cluster. In this approach, the emphasis is on similarities between structural features of call graphs.

2.4 Fingerprint Matching

After fingerprint generation, different fingerprint matching approaches can be applied in order to find similar fingerprints to a target function's fingerprint. These matching techniques include exact and inexact matching approaches. Figure 4 summarizes our proposed taxonomy for fingerprint matching.

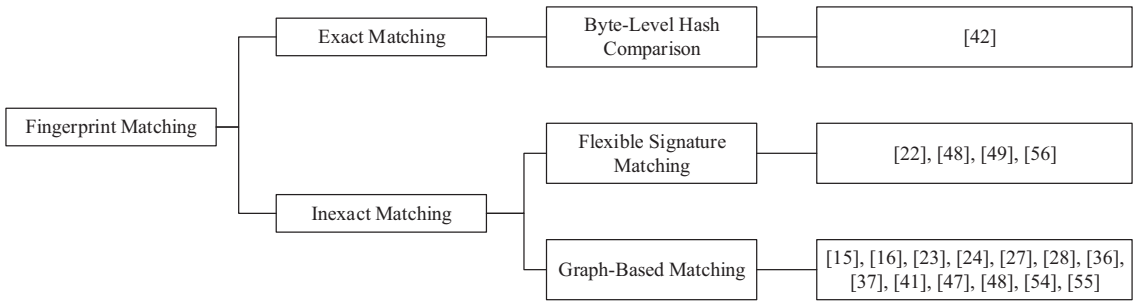


Figure 4: Fingerprint Matching Taxonomy

2.4.1 Exact Matching

A number of studies have been carried out that focus on exact data matching and apply byte-level sequence patterns to match function code. Some of these studies use exact matching for fingerprinting library functions in binary files (e.g., [42, 48, 77]). Even though exact pattern matching methods are accurate in detecting unmodified library functions, they are ineffective in case of slight byte-level discrepancies since they rely on byte-level sequence pattern matching. In practice, small changes to code results in different byte-level fingerprints in a way that can render wildcard patterns ineffective. Also, variation in source libraries and compilation settings introduce differences at byte-level representations.

Byte-Level Hash Comparison

The industry-standard disassembler IDA Pro is used by numerous reverse engineers [48, 55]. It provides a built-in capability for recognizing standard library functions and hex code sequences that are generated by common C-family compilers (e.g., statically linked library code, helper functions, and initialization code). The hash tree algorithm (described previously in Section 2.3.1) that implements this feature is part of a platform-independent technology known as IDA F.L.I.R.T (Fast Library Identification and Recognition Technology) [42]. IDA F.L.I.R.T verifies if each byte of the disassembled program can match the beginning of one of the standard library functions that can be detected.

Although F.L.I.R.T is a useful and efficient mechanism, it has certain limitations and is not necessarily robust when dealing with slightly obfuscated files. Library functions with small byte-level discrepancies are not recognized by IDA Pro since F.L.I.R.T merely holds exact function signatures. In addition, signatures generated from one version of a library may not be applicable to fingerprinting other versions of the same library. The inability to perform approximated or inexact matching is a major limitation of function fingerprinting in IDA Pro.

Moreover, this mechanism is designed merely for C/C++ programs and may not be functional for programs written in other languages. Also, it is only applicable to functions of the code segment and does not analyze functions of the data segment. In malware and obfuscated binaries, data bytes often hide code sequences.

The F.L.I.R.T mechanism may fail to fingerprint functions even in partially obfuscated binaries. Furthermore, F.L.I.R.T is only applicable to well-known compilers and their accompanying shared libraries. Despite prominent use of open source libraries in malware, F.L.I.R.T does not possess the knowledge (signatures) to fingerprint these well-known libraries. Overall, the efficiency and robustness of this mechanism can be significantly improved by employing more robust (syntactic and semantic) function fingerprinting techniques.

2.4.2 Inexact Matching

Contrary to exact matching approaches, other studies examine inexact matching approaches that include flexible signature matching techniques alongside graph-based analysis to measure the similarity between programs [22, 23, 37, 46, 48].

Flexible Signature Matching

As opposed to exact matching and byte-level hash comparison techniques, flexible signature matching techniques measure the similarity between programs while allowing for some byte-level discrepancies [22, 48]. For example, the work in [48] identifies indirect invocation of system calls (wrapper functions) through an inexact matching technique. Specifically, the approach presented in [48] considers three complications: function inlining, code reordering, and minor code variations. In order to address these complications, wrapper functions in a binary are identified through the process of flexible pattern matching. Functions are identified by means of a relaxed pattern matching technique in which inexact matches are permissible. When multiple

matches are found, all matching fingerprints are returned.

Instruction hash comparison is a simple, fast technique used to fingerprint and match functions and assembly instructions [49]. The hashing captures semantic information by being applied to a subset of function instructions and basic blocks. Locality sensitive hashing allows performing inexact matching on the fingerprints. It can be used to cluster similar functions. A hash value can represent the semantics of basic blocks. As mentioned previously, the work in [56] terms the semantic hash values being matched as semantic "juice".

Graph-Based Matching

One of the most well-known graph-based approaches to binary comparison is proposed in [37]. The BinDiff algorithm has inspired a number of works that use graph matching for similarity measurement, patch analysis, as well as malware detection (e.g., [15,16, 23, 24, 27, 28, 36, 41, 47, 48, 54, 55]). This approach treats binaries as directed graphs (generated from CG/CFG) and puts emphasis on structural similarity rather than the sequence of individual instructions. As a result of this abstraction, minor byte-level discrepancies (e.g., instruction reordering) can be tolerated when matching different versions of related programs. Besides, the graph-based approach is able to capture the flow and structure of programs more effectively. The similarity matching problem is formulated in this approach as a graph isomorphism problem. In this scheme, each basic block is abstracted as a tuple, which embodies the number of call graph nodes, edges, and function calls.

In order to find the similarity between two binaries, functions that share common signatures are grouped together. Then, using a more refined search process, the neighboring nodes of similar functions are examined to find potential matches. This solution is fast and effective in detection of differing code fragments in successive patched binaries and it performs best in isomorphic problems. However, this approach is not as effective when applied to non-isomorphic malware variants. Many malware variants (such as `Zeus` and `Citadel`) are built from the same core, but exhibit a completely different CG/CFG structure in spite of sharing a number of similar functions. In such cases, solutions that rely on approximated graph matching are more effective in pinpointing the similarity among binaries.

An explicit notion of similarity (or dissimilarity) enables us to quantify the level of similarity (or dissimilarity) between arbitrary binaries. Building upon BinDiff, Bourquin et al. developed BinSlayer using an approximate bipartite graph matching technique [23]. This approach benefits from a cost matrix and the Hungarian algorithm [5] that attempts to minimize the edit distance. The process of subgraph matching in BinSlayer entails three steps: (1) finding similar nodes with high degrees of confidence using a BinDiff-like approach and obtaining the set of unmatched functions as output, (2) applying the Hungarian algorithm to the unmatched nodes using a normalized cost function based on graph dissimilarity, and (3) checking the final mappings using a validator to ensure structural similarity and evaluate the quality of output matches. This process has been shown effective in accurate detection of similar nodes in binaries. However, one limiting factor of this approach in function fingerprinting is that it places all the attention on structural similarity, ignoring the

instruction semantics of each basic block. For example, if two functions perform similar operations using different control structures, this approach might not be able to match them. To further improve the fingerprint matching process, each fingerprint should take into account the types of instructions that occur in each basic block, as well as the structural information.

2.5 Fingerprinting Applications

The processes of function fingerprinting and matching have several applications and can be effective for many purposes. Figure 5 sums up our proposed taxonomy for the fingerprinting applications.

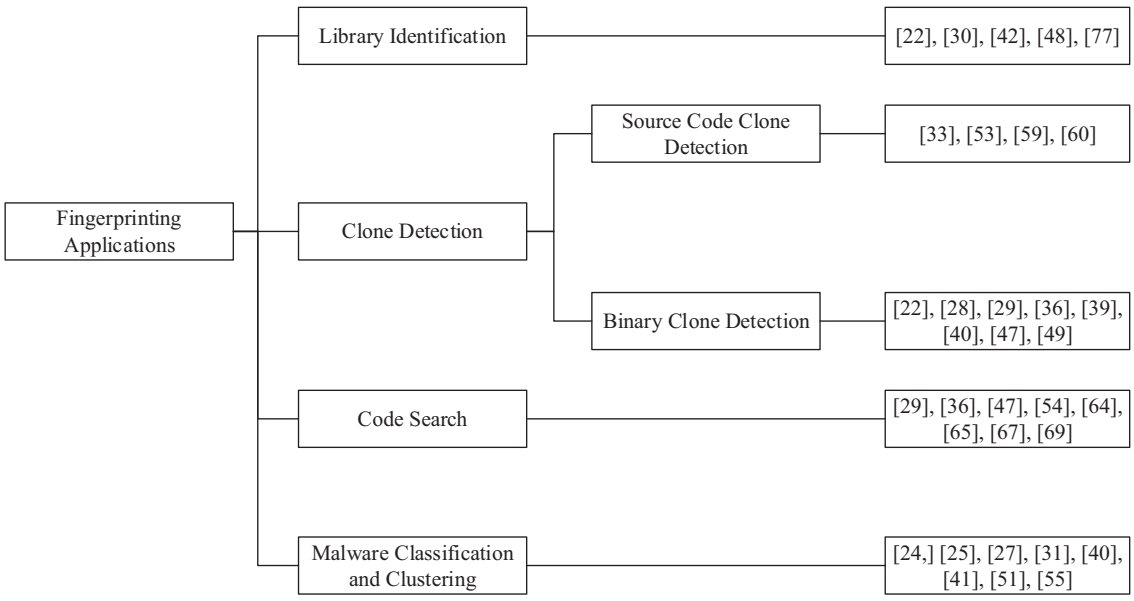


Figure 5: Fingerprinting Applications Taxonomy

2.5.1 Library Identification

The problem of library fingerprinting deals with the recognition and identification of target functions with regard to a set of known function patterns (signatures). The outcome of the library fingerprinting process is a set of labels (tags), assigned to target functions according to their identifiable features. The tags are important to malware analysts since they allow shifting the focus away from irrelevant functions and putting the spotlight on the payload. Besides, successful recognition of library functions help analysts understand the context of binary code faster and more accurately. It can also help reduce the number of functions that require manual inspection by the reverse engineer. In malware analysis, identification of cryptographic functions is an important step towards uncovering ciphers, understanding malware intents, and pinpointing encrypted (possibly stolen) data. Disassembly functions can be roughly categorized into three groups of: user functions, linked library functions, and imported library functions. One of the objectives of function fingerprinting is to distinguish between these functions automatically.

A number of studies have been carried out on fingerprinting library functions in binary files (e.g., [22,30,42,48,77]). Most of these techniques use byte-level sequence patterns to achieve exact data matching of function code. These exact pattern matching techniques are effective in detecting unmodified library functions. However, these techniques are rendered ineffective in the presence of any byte-level variations. Slight changes in the code can result in different byte-level fingerprints, which can cause the pattern matching techniques to become ineffective. Differences in the byte-level

representations can be a result of variations in the source code of the libraries or different compilation settings.

Other library identification studies examine flexible signature matching techniques. For instance, the work in [48] investigates the problem of linked library fingerprinting in Linux binaries and presents an inexact matching solution to identify indirect invocation of system calls (wrapper functions). The approach attempts to satisfy in a balanced manner two objectives: achieving high precision when labeling wrapper functions and delivering as much information as possible about the binary. This results in a tradeoff between unidentified functions and functions with multiple possible labels. The approach favors delivering several possible labels over an unidentified wrapper function. In malware analysis, fingerprinting wrapper functions, which often encapsulate system calls, is significant to the understanding of the context of the caller binary functions.

Such fingerprinting approaches are promising for detecting standard libraries, and demonstrate higher recall in comparison to exact matching techniques. However, solutions that rely merely on the identification of known (standard) function declarations (invocation interfaces) may not be directly applicable to fingerprinting other types of functions such as malware functions since malware authors typically modify those interfaces at source level to evade detection. Furthermore, solutions that perform symbolic execution are computationally expensive and often do not scale well.

2.5.2 Clone Detection

Clone detection is another relevant research area to binary component fingerprinting. Program components are built from smaller code fragments and building blocks, which might have originated from other sources such as open-source projects and shared libraries. Identifying the links between projects and recognizing shared program elements will enhance and accelerate the reverse engineering process. This problem can also be formulated as clone analysis and detection, and can be approached from various directions.

Source Code Clone Detection

The research works on source code clone detection investigates string-based, token-based, structure-based, and semantic-based approaches to detect four types of code clones [33, 53, 59]. In the simplest form, type I clones are known as exact clones. In type II clones, the syntactic structure is often preserved and the variation is related to slight changes in identifiers, layout, comments, and types. In a more realistic scenario, type III clones are defined to have altered code fragments. Type IV clones (also known as semantic clones) have different syntactic features with the same functional behavior.

Exact signature matching approaches can easily capture type I and II clones. However, detection of type III and IV clones may require more sophisticated fingerprinting algorithms. One important application of source code clone analysis is in plagiarism detection as presented in [60]. In that context, an accurate plagiarism detection

system should be robust to code transformations such as variable and parameter renaming, reordering of class attributes and methods, and slight changes in control flow and loop structures.

Binary Clone Detection

Applying source code clone detection techniques to disassemblies is not a straightforward process due to limited source-level information such as identifier names, typing information, function names, etc. The research on binary clone detection attempts to measure the similarity between binary files and identify clones using compiled code [22,28,29,36,39,40,47]. The syntactic approaches apply several steps of normalization to assembly instructions. Utilizing text search over identical byte sequences is common in detection algorithms that target type I clones. Combining token-based analysis with hash fingerprints and n-gram analyses can potentially improve detection of types II and III clones.

One important observation is that the development environment (i.e., the compiler, compilation configurations, language, IDE, etc) could leave specific fingerprints on output binaries. Two binaries that are compiled from similar source code using analogous compilation settings, tend to share some assembly functions. Combining static analysis with symbolic execution can improve clone detection results. The approach presented in [28] measures the provenance similarity of binaries based on the classification of semantic and syntactic features. Because this approach benefits from a symbolic execution component, it is able to reason about the dynamic effects of

instructions on the state of the program. Therefore, it is a good candidate for detecting clones of type III and type IV. The dynamic analysis techniques that monitor input/output behavior of functions result in lower false positive rates in detection of semantic (type IV) clones [49]. Morphological analysis and static code synchronization can constitute other possible solutions for binary clone detection [22]. In this approach, the paths of the CFG are reduced according to jump and call instructions. Identifying types III and IV clones are relevant applications of this approach. On the other hand, BinDiff-inspired subgraph search techniques are capable of detecting binary clones of types I-III.

2.5.3 Code Search

A closely related research direction models clone detection as a structural search problem and seeks to match low-level byte sequences with high-level source code patterns [67]. In contrast to binary clone detection techniques, this line of research attempts to measure the similarity between source and binary. It also aims to reveal linkage and provenance of binaries back to known source code and projects. In this approach, assembly-level features such as immediate values (constants), character sequences (strings), local and external function calls, and library imports are exploited to generate an approximated feature vector for detecting code reuse across shared and open-source libraries. Since code reuse is prevalent in malware development, the matching approach would be relevant to detection of types III and IV clones in common cryptographic and network-related libraries. Furthermore, this approach

is promising in fast reverse engineering of malware offspring with known predecessors [69].

Several works perform binary clone detection through a binary search engine that depends on certain key features of the binary function [29, 36, 47]. The approach presented in [47] proposes a binary search engine based on features extracted from the binary function's CFG. The approach employs the longest common subsequence algorithm on two different levels: the basic block level and execution path level. Next, neighborhood exploration is used on the basic blocks of the longest common path between the binary functions to further extend the mapping. Another assembly clone search engine is proposed in [36]. The proposed approach addresses the isomorphism problem for the assembly functions' CFGs. When attempting to match a target function, cloned subgraphs of functions in the dataset are identified. The work in [29] presents an approach that relies on a selective inlining technique. The proposed technique captures the semantics of the function by inlining relevant library and user-defined functions.

Leveraging a search engine for the purpose of binary analysis has various advantages for highlighting code reuse. As shown in [54], combining n-grams, n-perms, data constants, and control flow subgraphs to build composite models results in high recall and F2 rates for detecting binary functions. However, an efficient indexing mechanism is required for fast storage and retrieval of subgraphs. When combined with symbolic execution, this approach can be used for finding equivalent functions (Is-pairs), and potential matches (May-pairs) [64]. Expose uses the cosine distance

metric to compute the angle between two feature vectors representing a pair of functions. A distance score will be assigned to each pair of functions based on properties of neighbors of candidate May-pairs, and caller/callee relationships. Functions with common properties are placed in the same group and the group information is used for ranking.

In some of the aforementioned approaches, an intermediate representation is necessary to capture the semantics of code elements. Direct code comparison is computationally intensive and requires large amounts of storage and memory. More efficient approaches utilize indirect comparison techniques [65]. Overall, generating a compact intermediary representation for program elements makes it easy to fingerprint components and detect clones.

2.5.4 Malware Classification and Clustering

Binary code fingerprinting can also be applied in the field of malware detection, classification, and clustering. Several research works focused on extracting features and generating fingerprints (or signatures) for the purpose of malware classification or malware clustering (e.g., [24, 25, 27, 31, 41, 51, 55]).

The work in [27] proposes a malware fingerprinting process that can be used for malware variant detection. Features extracted from the set of CFGs contained in a piece of malware code is used to generate a fingerprint that describes that specific malware. The generated fingerprints are vector-based signatures. The evaluation of the proposed methodology shows that the approach is highly effective in detecting malware variants and has a low false positive rate. This approach could also be used

for clustering malware variants and grouping them with the appropriate malware family. In case numerous instances of a certain malware family are identified, then human analysis of that family might be required in order to study it and determine the influence that malware has.

Another malware detection technique that focuses on Android malware detection is presented in [41]. This method relies on features extracted from the call graph of a function. This approach utilizes the process of machine learning classification of graphs for the purpose of solving the problem of malware detection. Machine learning and feature analysis techniques are deployed for building a classification model that uses the extracted call graph features to categorize software code as either benign or malicious.

Additionally, a different malware classification approach based on call graph clustering is proposed in [55]. The proposed approach represents malware samples as call graphs. The idea is to apply generic detection techniques by clustering similar malware samples together. As a result, the focus is placed on the common features between the malware samples within a cluster. A number of different clustering algorithms are used to enable the detection of similar malware samples. Real malware samples are used to conduct several clustering experiments. The results of these experiments are evaluated against manual classifications received from human malware analysts. The conducted experiments convey that the process of clustering based on fingerprints generated from call graph features is an effective method for accurately detecting malware families.

Another proposal for malware detection is presented in [40]. Deterministic clone

detection approaches are introduced for the purpose of improving the recall rate of malware detection and assisting the malware analysis process. The proposed methods permit for performing exact and inexact matching at different token normalization levels. The signature generation process involves generating a binary vector out of the features, partitioning it into sub-vectors, then hashing the sub-vectors. The detection process is window-based, where each window contains a pre-defined number of instructions from the binary code. The proposed methodologies are evaluated through several experiments performed on real-life malware binaries.

2.6 Summary

We have presented different approaches for binary function fingerprinting and matching as well as the correlated problem of clone detection. A summary of the most relevant reviewed research works is presented in Table 2.

Different types of features can be combined together in the fingerprint generation process. Hashing fingerprinting techniques produce a compact fingerprint representation and allows for a fast comparison. Byte-level signature matching approaches are very sensitive to any minor changes in the binary function. Moreover, this kind of fingerprint only allows for exact matching. Graph-based fingerprinting however can tolerate minor byte level discrepancies by treating binaries as directed graphs and placing emphasis on structural similarity rather than the sequence of instructions. This solution is effective when detecting changed code fragments in successive patched binaries and isomorphic problems. Nonetheless, this approach is not as effective when

Reference	Binary Framework	Similarity Metrics	Matching Algorithm
[56]	BinJuice	Semantic hashing through symbolic evaluation	Structural comparison, hash comparison
[49]	-	Semantic hashing	Min-hashing
[42]	IDA Pro F.L.I.R.T.	Pattern hashing	Byte-sequence matching
[37]	BinDiff	Node properties	Heuristic CFG matching
[27]	-	CFG sub-graph properties	Vector clustering, vantage point trees
[41]	-	Structural call graph properties	Feature map clustering
[55]	-	Structural call graph properties	Clustering, graph edit distance
[48]	Unstrip	Semantic descriptors of wrapper functions	Flexible pattern matching
[23]	BinSlayer	Node properties	Hungarian algorithm
[40]	BinClone	Hash of opcodes	Window-based, syntactic matching
[47]	BinSequence	CFG properties	Longest common sub-sequence algorithm
[36]	Kam1n0	CFG sub-graph properties	Sub-graph isomorphism search

Table 2: Summary of Binary Analysis Frameworks for Function Fingerprinting

applied to non-isomorphic variants. It can also be relatively expensive to compare large graphs. The reviewed literature inspired us to construct a hybrid fingerprinting approach that considers the information provided by the different types of features (syntactic, semantic, and structural) while taking advantage of the compactness and efficiency of hashing techniques. After describing the advantages and shortcomings of each approach, the following chapter elaborates the design and implementation details of the BinSign framework.

Chapter 3

A Framework for Fingerprint Generation and Matching

3.1 Overview

An effective fingerprinting approach should produce a unique and compact representation of the functionality of a binary function. Functions that perform similar functionalities should be assigned similar fingerprints. In addition, it must be robust to byte-level discrepancies and tolerant of differences in the structure of basic blocks, call graphs, and control-flow graphs. Moreover, the probability of a fingerprint collision must be negligible. In this thesis, we design BinSign, a fingerprinting framework for binary functions. The proposed approach consists of two main components: (1) Scalable fingerprint generation and indexing of a large dataset, (2) Fingerprint matching to detect known signatures in target binaries. Our fingerprint generation approach relies on a set of features that are extracted from assembly functions (global

features). These features are combined with structural information from the CFG of a function by dividing the CFG into tracelets that carry semantic information while allowing the approach to be more scalable than an approach that considers all CFG paths. As such, BinSign fingerprints capture not only the syntactic information of a function, but also its semantics and underlying structure. Figure 6 depicts a high-level overview of the BinSign system.

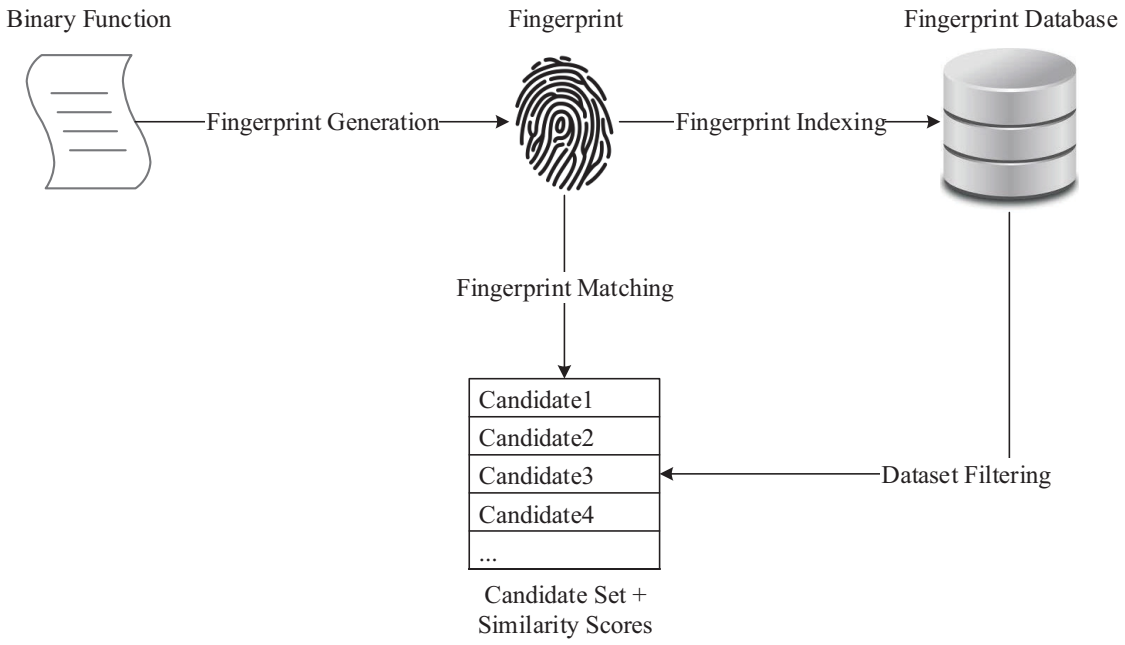


Figure 6: BinSign System

In order to achieve a great scalability in fingerprint matching, we design and implement an efficient fingerprint matching framework to match a target fingerprint against a large repository of fingerprints. With the purpose of avoiding pairwise comparison of a large volume of fingerprints, we use three mechanisms to facilitate achieving a scalable matching process. First, we leverage Locality-Sensitive Hashing (LSH) [58]

combined with min-hashing [17] for selecting fingerprint candidates. Second, we apply a filter to further prune the search space based on the number of basic blocks of the CFG. It is highly unlikely for a pair of functions to match if the difference in the number of their basic blocks is significant. Finally, BinSign system's computation is distributed on multiple machines using Rabbit MQ [9] in order to further improve the performance and scalability. In addition, BinSign framework computes similarity scores between the target function and each selected candidate using Jaccard similarity. To improve the accuracy of the matching process, features are ranked according to their significance while calculating the similarity score of two fingerprints.

3.2 Threat Model

3.2.1 In-Scope Threats

We design BinSign for the purpose of facilitating the reverse engineering process and helping the reverse engineer throughout the process. By matching binary function fingerprints, BinSign can assist in multiple processes associated with reverse engineering including compiler identification, clone detection, library function identification, code similarity, malware detection and classification, provenance analysis, vulnerability detection, and authorship attribution among other uses. BinSign is designed to be resilient to changes in the binary code such as those introduced into the code by the use of different compilers and selecting different optimization levels when compiling the code. Moreover, the fingerprinting and matching techniques are also designed to be resilient to any light obfuscation methods that do not alter the structure of the

function's CFG and binary instructions in its code in a major manner. These light obfuscation techniques include register replacement, register reassignment, dead-code insertion, and code substitution among others. Name stripping and removal of symbolic information for instance are other factors that do not affect our fingerprinting and matching processes.

3.2.2 Out-of-Scope Threats

The use of BinSign as a tool is not intended for replacing the reverse engineering process entirely, but merely to assist and support it. Some binary code, such as malware, is often packed or encrypted to avoid inspection by anti-virus software. In order to analyze and examine such pieces of code, various unpacking techniques can be used [19,32]. Obfuscation methods also pose a challenge against reverse engineers who attempt to analyze an obfuscated piece of code. Automated tools have been developed that implement several de-obfuscation techniques [71,76]. However, unpacking and de-obfuscating a piece of code lie outside the scope of our considered threat model. We assume the binary code is already unpacked and de-obfuscated before going through the proposed fingerprint generation or matching processes. Moreover, our fingerprinting methodology is not designed to be resilient to any heavy obfuscation techniques, which result in significant modifications in the structure of a function's CFG or heavily changed instructions of a binary function. Therefore, a function would not be successfully matched after going through heavy obfuscation (such as control-flow flattening or extensive instruction substitution).

3.3 Feature Extraction

A fingerprint provides valuable insights regarding a binary piece of code. These insights should not only capture the syntactic elements of a code fragment but should also capture its semantics and underlying structure. Thus, it is important to select the suitable set of assembly features that characterizes the essential semantics and functionality of programs. Some factors such as name stripping and removal of symbolic information can eliminate access to some features thus complicating the matching process.

Indeed, some features such as strings and function names are unavailable in stripped binary files. In order to overcome these complications, the semantics of code operations are captured by analyzing mnemonic groups and operand types even if the symbols are stripped. Feature-driven fingerprint generation entails several steps of feature extraction and normalization. Each function's fingerprint includes a feature vector $\vec{v}_{f_i} \in \vec{V}$ that captures all the available features of a function f_i in the form of key/value pairs (k_i, v_i). Choosing the right combination of features that carry enough code semantics has a direct impact on the fingerprinting results and fingerprint matching accuracy. In what follows, we present the features considered by BinSign's fingerprint.

The described features are extracted at two different levels from a binary function, namely global features and tracelet features. Features that describe each individual basic block such as the instructions and constants that appear in the basic block are extracted from each block and then combined together to form the tracelet features.

The structure of the CFG is captured in the function's fingerprint through these tracelet features. Table 3 depicts the list of extracted tracelet features. However, features that occur once per function and describe the function as a whole (such as the return type, number of arguments, and function size for example) constitute the global features. The extracted global features are listed in Table 4. The symbols "#" and "∗" denote "number of" and "size of" features, respectively. Some features are considered to be common to both tracelet features and global features as well. For instance, the number of instructions in each basic block is extracted as a tracelet feature and the total number of instructions of the function is extracted as a global feature.

Data Constants	`Constants`
	`Strings`
	`#Constants`
	`#Strings`

Tracelet Info.	`#Instructions`
	`#Operands`
	`Code Refs.`
	`#Code Refs.`
	`Function Calls`
	`#Function Calls`
	`Imported Functions`
	`#Imported Functions`

Functionality Tags	`#API Tags`
	`#Library Tags`
	`#Mnemonic Groups`

Table 3: Tracelet Features

Data Constants	#Constants #Strings
Prototypes	Return Type Arguments #Arguments *Arguments
Function Info.	#Instructions *Local Variables Function Flags #Code Refs. #Function Calls #Imported Functions *Function #Basic Blocks Tracelets #Tracelets
Functionality Tags	#API Tags #Library Tags #Mnemonic Groups

Table 4: Global Features

During the feature extraction process, functions are examined in different layers. Each group of features encapsulates lower level features into higher levels of abstractions. We take into consideration the following groups of information, each of which describes a different aspect of an assembly function during the feature extraction and fingerprint generation processes. In the following, we describe the groups of features in more detail.

3.3.1 Characterization of Function Prototype

Each function's prototype carries valuable information about the function, such as the return type, and the number, size, and types of arguments. Therefore, we consider this information provided by the prototype in the feature extraction and fingerprint generation processes.

3.3.2 Composition of CFG Instructions

This group of features captures information describing the CFG's structure and instructions in its basic blocks such as the number of basic blocks, number of tracelets, the number and types of instructions that appear in each basic block. The mnemonic and operand features of each instruction are extracted and normalized.

The normalized mnemonics list contains a generalized representation of assembly instructions, with the operands numbered according to their types. For instance, general registers ($reg : eax...edx$) are replaced with 1, memory references (mem) are coded as 2, immediate values (imm) are replaced with 5, and so on. The mnemonics list is then reduced to a compact form following a simple frequency analysis: the number of occurrences of each assembly instruction and the total number of calls to registers and memory addresses are determined.

Following this step, the instruction mnemonics are classified into fifteen groups according to their operation group. Table 5 lists these mnemonic groups and the mnemonics that belong to each of them. After that, the total number of instructions and the distinct number of instructions of each mnemonic group are computed. Any data constants and strings that occur in these instructions as operands are also captured in the fingerprint as well as the local variables.

Mnemonic Group	Mnemonics
Data Transfer	mov, xchg, cmpxchg, movz, movzx, movs, movsx, movsb, movsw
Data Comparison	cmp, cpx, cpy, test, cmn, teq, tst
Logical Operations	xor, or, and, not
Stack Operations	push, pop, pushf, popf, pusha, popa, pushad, popad
Flag Manipulation	sti, cli , std, cld, stc, clc, cmc, sahf, lahf, popfw, popf, popflq, popfl, pushfw, pushf, pushflq, pushfl
Binary Arithmetic	sub, add, inc, dec, mul, div, shl, sal, sar, shr, ror, rcl, rcr, rol
Control Transfer	jz, jnz, je, jne, jg, jge, ja, jae, jl, jle, jb, jbe, jo, jno, jc, jnc, js, jns, jecxz
Floating Operations	fbld, fbstp, fcmovb, fcmovbe, fcmove, fcmovnb, fcmovnbe, fcmovne, fcmovnu, fcmovu, fild, fist, fistp, fld, fst, fstp, fxch, fabs, fadd, faddp, fchs, fdiv, fdivp, fdivr, fdivrp, fiadd, fidiv, fidivr, fimul, fisub, fisubr, fmul, fmulp, fprem, fprem1, frndint, fscale, fsqrt, fsub, fsubp, fsubr, fsubrp, fxtract, fcom, fcomi, fcomip, fcomp, fcompp, ficom, ficomp, ftst, fucom, fucomi, fucomip, fucomp, fucompp, fxam, f2xm1, fcos, fpatan, fptan, fsin, fsincos, fyl2x, fyl2xp1, fld1, fldl2e, fldl2t, fldlg2, fldln2, fldpi, fldz, fclex, fdecstp, ffree, fincstp, finit, fldcw, fldenv, fnclex, fninit, fnop, fnsave, fnstcw, fnstenv, fnstsw, frstor, fsave, fstcw, fstenv, fstsw, fwait
String Manipulation	cmps, cmpsq, cmpsb, cmpsd, cmpsl, cmpsw, lods, lodsq, lodsb, lodsl, lodsd, lodsw, movs, movsq, movsb, movsl, movsd, smovl, movsw, smovw, scas, scasq, scasb, scasl, scasd, scasw, stos, stosq, stosb, stosl, stosd, stosw
Repeat Operations	rep, repe, repz, repne, repnz
Call Operation	call
Jump Operation	jmp
Halt Operation	hlt
Load Operations	lea, les
Interrupt System	Mnemonics with sys_ prefix

Table 5: Mnemonic Groups

3.3.3 Types of Local, System, and API Calls

The number and types of function calls that take place in the code of a binary function are other features we consider. System and API calls are important features that should be taken into account for inexact pattern matching since they are good indicators of the functionality of the code. Examples of API Categories are shown in Table 6.

Each function can be categorized according to its execution outcome on a target system. System calls are considered as interaction points with the operating system and provide valuable information on the potential runtime behavior and functionality of the function. These system and API calls are used to assign functionality tags to the function.

A functionality tag is a code annotation that is assigned to code fragments in a function and provides a high-level description of the code context and side effects. Functionality tags are useful for fast identification and localization of specific groups of operations in disassemblies. Assigning context-based tags to assembly level functions provides many benefits when statically analyzing the binary function. Functionality tags provide hints to the reverse engineer about the potential computations that may take place in a certain function. As described in [69], a function is assigned multiple tags if it encloses a number of system calls. When combined with cross referencing of code and data, the tags can be used for semantic code search and tracking code functionality.

For instance, finding crypto-related keys is an important step in malware reverse

engineering. Hence, automatic assignment of crypto-related functionality tags (e.g. CRY, HSH, CER) to disassembly functions could limit the search space, reduce the amount of manual work, and highlight the most likely code fragments to look for keys. Moreover, combinations of functionality tags describe the overall functionality of the code regions and highlight the sequence of actions carried out for performing them (e.g., CRY+FIL+NET is translated into crypto-operations on a file, followed by network communication).

Category	**API Functions**
File	CreateFileMapping, GetFileAttributes, ReplaceFile
Network	gethostbyname, getaddrinfo, recv, WSAAccept
Registry	RegCreateKey, RegQueryValue, SHRegSetPath
Crypto	CryptGenKey, CryptSetKey, CryptDecodeObject
Service	QueryServiceLockStatus, SetServiceObjectSecurity
Memory	VirtualAlloc, VirtualUnlock, ReadProcessMemory

Table 6: Examples of API Categories

3.4 Fingerprint Generation

This sub-section is dedicated to describing the details of the steps involved in the fingerprint generation process. Our approach for generating binary function fingerprints is depicted in Figure 7. The approach consists of four steps: (1) Disassembling the binary file and extracting the function CFG, (2) tracelet generation from function CFG, (3) global feature and tracelet feature extraction, and (4) feature min-hashing.

In the following, we describe each of these steps in more detail.

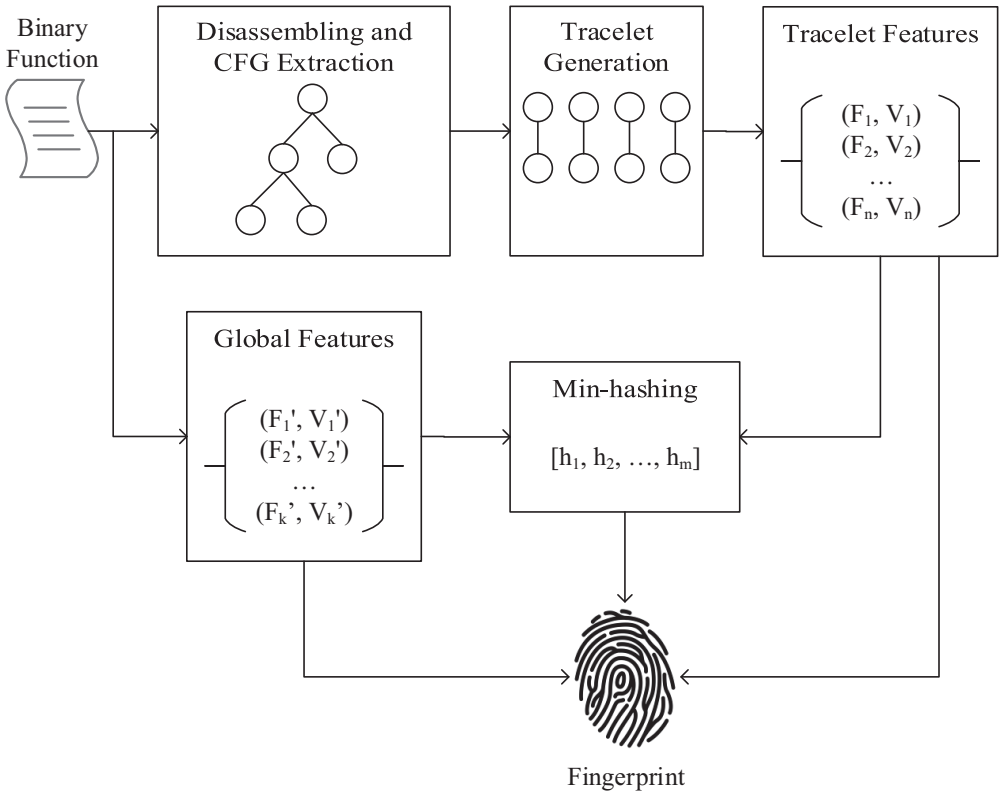


Figure 7: Fingerprint Generation Process

3.4.1 Disassembling and CFG Extraction

A significant aspect of our proposed methodology is related to generating fingerprints for functions in the binary format. To this end, we introduce a function fingerprinting approach for the generation and matching of abstracted representations from assembly functions. This approach enables us to fingerprint structural and syntactic function properties. Given a binary file, the first step is to disassemble it using the industry-standard disassembler *IDA Pro* [4]. Each disassembled binary executable file contains a set of assembly functions. In order to capture the structural information of a

function, we take into consideration its CFG. Each node in a CFG of a function represents a basic block and each edge denotes a branching instruction that results in a jump from one basic block in the CFG to another. CFGs are important because they capture syntactic elements (assembly instructions in basic blocks) and relationships (jumps/calls) between basic blocks.

3.4.2 Tracelet Generation

As mentioned previously, one of the main objectives of BinSign is to generate fingerprints that capture not only the syntactic information of a function, but also its core CFG structure. To this end, we decompose the CFG into traces of execution and extract features from the CFG tracelets. The intent is to capture all possible execution traces of a function. However, extracting all possible paths from a CFG is computationally expensive, especially in the case of very large functions. Moreover, it would be redundant and space consuming to consider the information captured from repeated nodes that are common between different paths multiple times.

In order to counter these issues, we adopt the idea presented in [35], by decomposing function CFGs into partial traces of execution, namely tracelets. In [35], tracelets are used after the instructions are put through a simple rewriting engine for the purpose of code search in order to find similar functions in a code base.

For our purposes however, we use the tracelets as part of the fingerprint that represents a binary function. Each tracelet is only considered once in the fingerprint and is not repeated. In BinSign, tracelets are comprised of two basic blocks. This means that each edge in the CFG represents one tracelet. Considering a larger number

of basic blocks per tracelet would result in redundant information without offering any additional benefits. This solution allows us to generate execution traces more efficiently and without losing in functionality since all CFG nodes and edges are visited at least once and the information they provide is captured in the fingerprint.

For example, consider the CFG presented in Figure 8. The CFG consists of five basic blocks that are connected by means of six edges. Therefore, information describing six distinct tracelets would be included in the fingerprint of the binary function represented by this CFG. Following the labeling of the basic blocks as in Figure 8, the generated tracelets can be summarized in a set of pairs of basic blocks as follows: (A, B), (A, E), (B, C), (C, C), (C, D), (D, E). It is worth noting that the CFG contains a loop at the basic block labeled as C. The loop is traversed only once when decomposing the CFG into a set of tracelets.

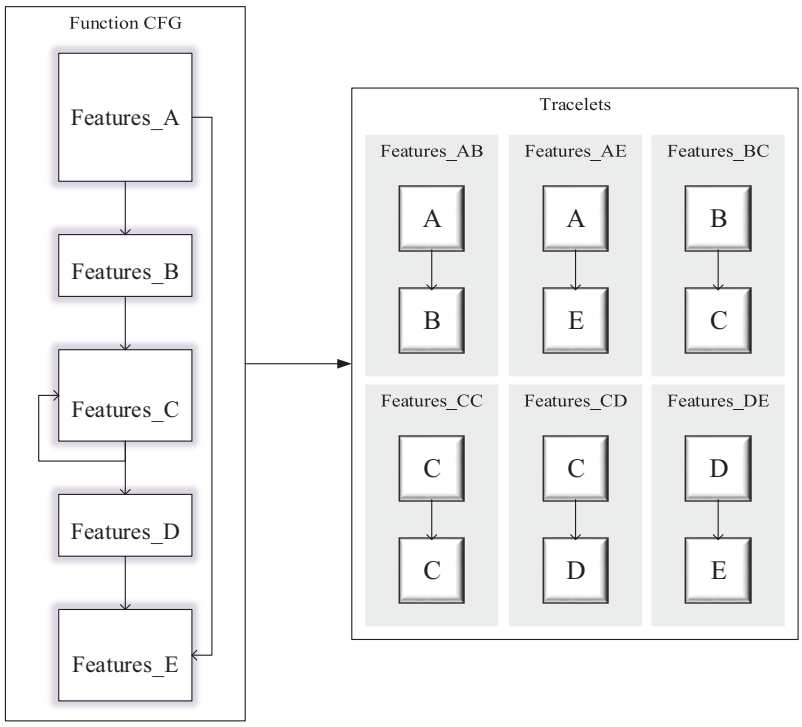


Figure 8: Example of Tracelet Generation

3.4.3 Feature Extraction

After disassembling the function, extracting its CFG, and decomposing the CFG into tracelets, the features previously described in Section 3.3 are extracted. These features are then combined to form the global features and tracelet features. The global features are extracted once from the function as a whole, and the tracelet features are extracted from the basic blocks of each tracelet. The collection of these features compose a descriptive fingerprint that carry a lot of valuable information about the binary function.

3.4.4 Signature Hashing

We take advantage of the *min-hashing* technique [17] in order to produce a compact representation of the signature. In essence, min-hashing is a technique that is used to reduce the dimensionality of a set of values using a number of hash functions. Algorithm 1 describes the process we use for generating the signature hash. It takes, as input, an assembly function and produces, as output, the hash of the function's signature.

It is important to mention that the min-hashing process is not applicable to all the features being extracted, i.e., to the numerical features. Therefore, we do not apply min-hashing to all the extracted features. Instead, min-hashing is applied to the normalized instructions of the assembly function. The function's normalized instructions constitute a suitable representation of the binary function's functionality and can be used effectively later on to filter the functions in the dataset during the fingerprint matching process to acquire a candidate set of fingerprints that are most

similar to the target function's fingerprint.

Each instruction in the function's binary code is normalized by replacing the operands with a number that represents the operand's type. After normalizing the instructions of a binary function, each hash function is used to hash all normalized instructions and the minimum hash value is selected. Multiple random numbers (seeds) are used in combination with the md5 hash function in order to generate many different hash functions for the min-hashing process. The seeds are generated randomly beforehand, and remain constant every time a new fingerprint is generated. This is necessary for the signature hashes of different fingerprints to be comparable to each other.

After conducting several experiments, we select 210 hash functions to be used in the min-hashing process to generate the signature. Our experiments show that this is a suitable number of hash functions to generate an effective signature. The collection of the minimum hash values resulting from hashing the normalized instructions using all the hash functions represents the signature hash. In other words, the signature hash consists of a vector of m min-hash values, where m is the number of hash functions. In Section 3.5, we describe how this min-hash signature is used in the fingerprint matching process in order to filter the functions in the dataset and acquire a candidate set.

Algorithm 1: Generating a Signature Hash

Input: An assembly function

Output: A signature hash

Function *Generate_Signature_Hash(function)*

```
minHashValues = [ ]
graph = getCFG(function)
normalized_instructions = graph.getNormalizedInstructions()
/* Get the min-hash values of the instructions                    */
/* seed1 and seed2 are previously generated random numbers        */
for (seed1, seed2) in getRandoms() do
    /* For each hash function                                     */
    hashValues = [ ]
    for instruction in normalized_instructions do
        /* Generate a hash value for instruction from hash functions
           by combining md5 with two random numbers               */
        hashvalue = generateHashValue(md5(instruction), seed1, seed2)
        hashValues.append(hashvalue)
    end
    /* Append the minimum hash value                              */
    minHashValues.append(min(hashValues))
end
return minHashValues
end function
```

Instructions	HashFunction_1	HashFunction_2	...	HashFunction_m
Inst_1	123	2	...	705
Inst_2	294	946	...	60
Inst_3	874	212	...	381
Inst_4	42	100	...	529
Inst_5	509	87	...	91
Inst_6	15	446	...	173
...	...	...	...	...
Inst_n	39	568	...	446

Table 7: Example of Signature Hash Generation

Table 7 shows an example of the process of deriving the hash signature of a function from its normalized instructions. Each row in the table represents one normalized instruction and each column represents one hash function derived from the hash function *md5* using different seeds. Each cell in the table holds a hash value resulting from hashing the normalized instruction in the left most cell of the row using the hash function at the top of the column. The number of instructions in this example is denoted by n and the number of hash functions is denoted by m. In order to generate the signature hash for this example, the minimum hash value from each column is selected. Therefore, the signature hash can be represented through the following list of hash values: [15, 2, ..., 60]. This list would be of length m.

3.4.5 Fingerprint Components

The fingerprint generated through BinSign's methodology is composed of three different parts. As depicted in Figure 7, the parts that make up a fingerprint in BinSign are: (1) the function global features, (2) the tracelet features, and (3) the signature hash of the normalized instructions. Having different types of features as part of

the fingerprint allows the resulting fingerprint to carry various information about the syntax, semantics, structure, and functionality of a binary function.

The signature hash component of the fingerprint is mainly used in the fingerprint matching process in order to filter all the fingerprints in the dataset and acquire a candidate set. The similarity score is then calculated between each candidate fingerprint in the candidate set and the target fingerprint based on how similar the function global features and tracelet features are. Moreover, BinSign fingerprints can additionally be used to share information about a binary function without sharing the function's actual binary code through merely sharing the fingerprint. The normalized instructions are only used to generate the signature hash values without being stored as part of the fingerprint. Therefore, the instructions are not leaked when sharing a function's fingerprint. After generating the fingerprints, the functions are indexed in parallel into a `Cassandra` database. The scalability of the indexing process is discussed in Section 4.4.

In order to show an example of a function's fingerprint, Figure 9 depicts the CFG of a sample function. The details of the fingerprint of this function is shown in Table 8. In this example, features with the value 0 and empty lists are not included in Table 8 for simplicity purposes. In the tracelet features, the features of each basic block are enclosed between the symbols "{" and "}". Each tracelet includes two basic blocks, which are separated by the symbol "-".

Global Features	[(function_name, sub_5A4C2580), (basic_blocks_number, 5), (function_size, 65), (argument_size,8), (argument_number,2), (local_variables_size, 4), (flags, 17424), (arguments, [dword, dword]), (return_type, None), (instructions_number, 29)]
Tracelet Features	{instructions_number: 6, constants: [0L], mnemonic_groups: ['Stack', 'DataTransfer', 'Stack', 'DataTransfer', 'Jump', 'DataTransfer'], constants_number: 1}, {instructions_number: 3, constants: [32L] , mnemonic_groups: ['Compare', 'Branch', 'DataTransfer'], constants_number: 1} - {instructions_number: 4, constants: [1L, 32L], mnemonic_groups: ['DataTransfer', 'Arithmetic', 'DataTransfer', 'Compare'], constants_number: 2}, {instructions_number: 3, constants: [32L], mnemonic_groups: ['Compare', 'Branch', 'DataTransfer'], constants_number: 1} - {instructions_number: 3, constants: [32L], mnemonic_groups: ['Compare', 'Branch', 'DataTransfer'], constants_number: 1}, {instructions_number: 13, constants: [8L], calls: ['sub_5A4C2540'], mnemonic_groups: ['DataTransfer', 'DataTransfer', 'DataTransfer', 'Stack', 'DataTransfer', 'Stack', 'Call', 'Arithmetic', 'DataTransfer', 'DataTransfer', 'DataTransfer', 'Jump', 'DataTransfer'], constants_number: 1, calls_number: 1} - {instructions_number: 3, constants: [32L], mnemonic_groups: ['Compare', 'Branch', 'DataTransfer'], constants_number: 1}, {instructions_number: 3, mnemonic_groups: ['DataTransfer', 'Stack']} - {instructions_number: 13, constants: [8L], calls: ['sub_5A4C2540'], mnemonic_groups: ['DataTransfer', 'DataTransfer', 'DataTransfer', 'Stack', 'DataTransfer', 'Stack', 'Call', 'Arithmetic', 'DataTransfer', 'DataTransfer', 'DataTransfer', 'Jump', 'DataTransfer'], constants_number: 1, calls_number: 1}, {instructions_number: 4, constants: [1L, 32L], mnemonic_groups: ['DataTransfer', 'Arithmetic', 'DataTransfer', 'Compare'], constants_number: 2}
Signature Hash	[6685L, 643L, 7535L, 462L, 6978L, 14480L, 2965L, 3813L, 10682L, 1184L, 4993L, 21866L, 4582L, 19074L, 9137L, 694L, 4819L, 4939L, 7646L, 10449L, 2242L, 9081L, 5877L, 2914L, 3766L, 1061L, 3674L, 87L, 7301L, 13164L, 16519L, 7426L, 11339L, 5366L, 1024L, 6416L, 8080L, 5980L, 3931L, 41L, 5920L, 12543L, 10032L, 19143L, 4521L, 667L, 17382L, 630L, 3476L, 6095L, 4708L, 4666L, 394L, 2075L, 6405L, 15590L, 12420L, 4866L, 15238L, 9420L, 18267L, 500L, 6134L, 11105L, 2414L, 10262L, 14855L, 6275L, 9454L, 702L, 5986L, 7724L, 1326L, 3316L, 3595L, 5039L, 4919L, 13202L, 6449L, 581L, 10544L, 12750L, 558L, 9293L, 7255L, 3047L, 6526L, 2666L, 15273L, 3382L, 7677L, 9423L, 6666L, 8875L, 2222L, 19026L, 558L, 1534L, 5743L, 2865L, 14064L, 3086L, 6723L, 3701L, 6408L, 3412L, 23757L, 1205L, 2170L, 322L, 17558L, 6017L, 2229L, 3528L, 12931L, 1912L, 11654L, 4350L, 7752L, 4061L, 154L, 1108L, 1755L, 5416L, 16351L, 3334L, 7483L, 614L, 2818L, 1800L, 2185L, 12577L, 3171L, 4091L, 5375L, 6826L, 18492L, 11674L, 1332L, 521L, 9078L, 4934L, 3832L, 7271L, 18534L, 14107L, 468L, 731L, 9682L, 352L, 8749L, 14527L, 4195L, 13255L, 7341L, 2252L, 1526L, 4323L, 8914L, 4140L, 990L, 510L, 1427L, 18539L, 313L, 10923L, 2368L, 16289L, 2908L, 3866L, 3312L, 1799L, 4379L, 18681L, 6426L, 15823L, 3940L, 4604L, 565L, 100L, 863L, 4734L, 2423L, 1057L, 7348L, 11822L, 8594L, 838L, 7281L, 4429L, 184L, 5673L, 409L, 21617L, 18332L, 18251L, 11619L, 5693L, 2711L, 7436L, 880L, 14349L, 18036L, 8601L, 3447L, 6277L, 1848L, 7638L, 8870L, 12025L]

Table 8: Example of Function's Fingerprint

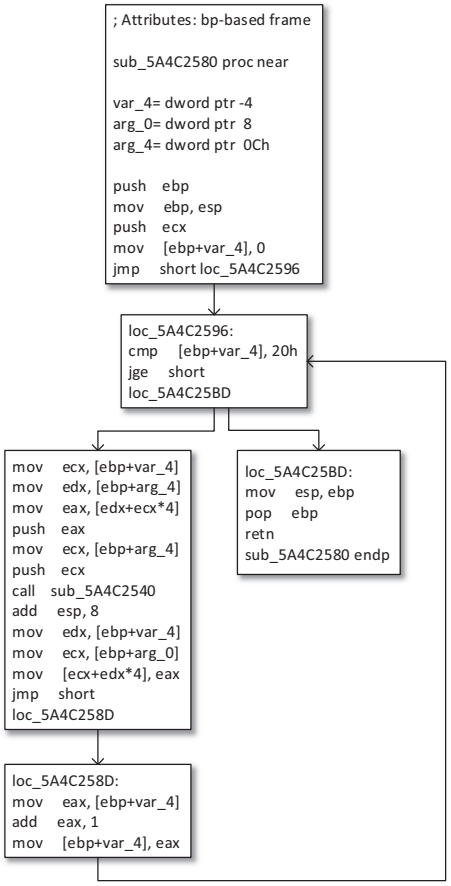


Figure 9: Sample Function's CFG

3.5 Fingerprint Matching

The fingerprint matching process aims at recognizing target functions and identifying functions of a target disassembly that share similar features with fingerprints of other functions in the dataset. Figure 10 depicts an overview of BinSign's matching process. Through the fingerprint matching process, a candidate set is selected from the entire dataset as being similar to the target fingerprint. After that, the similarity between each candidate fingerprint and the target fingerprint is computed. The fingerprints whose similarity lie above a certain threshold are then selected.

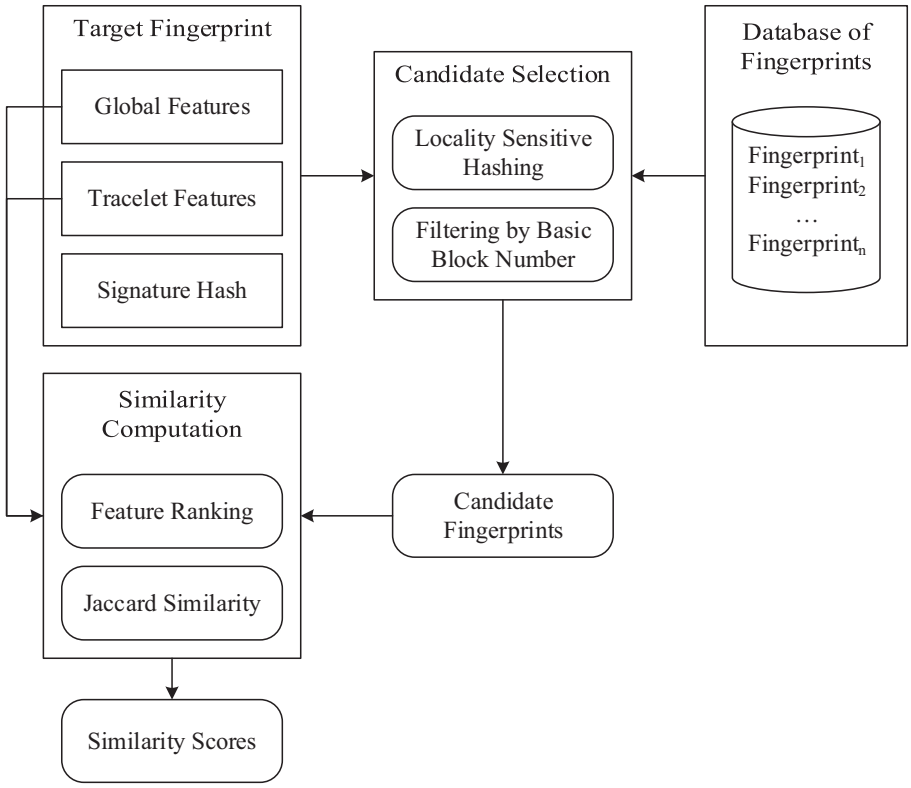


Figure 10: Fingerprint Matching Process

3.5.1 Fingerprint Candidate Selection

In order to select a candidate set when matching a function's fingerprint and reduce false positives, two filtering methods are implemented: filtering based on the number of basic blocks and LSH. First, the functions in the dataset are filtered by the number of basic blocks in the function's CFG. We choose this filter because it is unlikely for a function with a very small number of basic blocks to match a function with a significantly larger number of basic blocks. Accordingly, we set a threshold that is proportional to a function's number of basic blocks. More precisely, if the difference in the number of basic blocks between the target fingerprint and a fingerprint in the dataset is larger than 30% of the number of basic blocks of the target fingerprint, then

the candidate function is filtered out and it is not considered as a candidate match.

Second, LSH [58] is used in order to reduce the number of functions being considered. The idea consists of dividing the hash values of the min-hash signature into bands (Figure 11), where each band consists of multiple hash values. A fingerprint from the dataset is considered as a candidate match to a target function if all its hash values match the hash values of the target signature in at least one of the bands. The values of each band in the min-hash signature of the target fingerprint is used to create a query to obtain candidate similar functions from the database. The results of all the queries from all bands are then combined into the candidate set. For instance, in the illustrating example depicted in Figure 11, $Signature_1$ and $Signature_n$ are selected as candidates since all the hash values of at least one of their bands ($Band_2$ of $Signature_1$ and $Band_1$ of $Signature_n$) match the hash values of the corresponding band in the target signature.

In order to reduce the number of false positives, we conduct many experiments to choose the best number and size of bands. The smaller the band size, the more inclusive the candidate set, which leads to more false positives. However, if the band size is too large, true matches with some modifications will not be included in the candidate set. We find through experiments that a band size of *seven* hash values and a total of *thirty* bands constitute a suitable signature. This means that 210 different hash functions are used in the min-hashing process to generate the signature. Since LSH approximates the Jaccard similarity, this choice of band size and number of bands sets the similarity threshold to about 60%. According to [58] an approximation of the threshold is calculated as $(\frac{1}{\mathrm{b}})^{\frac{1}{\mathrm{r}}}$, where b represents the number of bands and

r represents the size of each band. Therefore, fingerprints with a similarity score between the normalized instructions of less than 60% are not included in the candidate set of similar functions.

		Target Signature	$Signature_1$	$Signature_2$	...	$Signature_n$
$Band_1$	$Hash_1$	1465	4486	145		**1465**
	$Hash_2$	614	614	56		**614**
	$Hash_3$	60	1715	478		**60**
$Band_2$	$Hash_4$	945	**945**	8675		45
	$Hash_5$	9247	**9247**	601		8965
	$Hash_6$	4559	**4559**	23775		2237
	...					
$Band_x$	$Hash_{m-2}$	605	1908	12560		9458
	$Hash_{m-1}$	190	25	7206		16759
	$Hash_m$	3127	999	189		15

Figure 11: Example of Candidate Selection

3.5.2 Fingerprint Similarity Computation

When matching a function's fingerprint, we need to calculate the similarity between a pair of functions. Hence, a similarity score between the target fingerprint and each fingerprint in the candidate set is calculated. The Jaccard similarity coefficient is used in order to calculate the similarity score between a pair of fingerprints. The Jaccard similarity between the global features of the target fingerprint and the candidate fingerprint is calculated then combined with the Jaccard similarity between the tracelets' features of both fingerprints. The combined Jaccard similarity coefficient of the global features and tracelet features is then considered to be the similarity score of the two fingerprints.

Each feature in the fingerprint has a different effect on the resulting similarity

score since some features are more significant than others. Therefore, each feature that influences the similarity score has a different weight as displayed in Table 9. The weights of the features are assigned after using Weka [12] for ranking the features. Weka is a tool that offers implementations of several machine learning algorithms. The gain ratio attribute evaluation algorithm [52] is used through Weka in order to rank the different features and assign weights to the features accordingly. This is performed through a supervised machine learning process where the names of the functions are known. All the considered features are extracted from each function in the dataset before ranking the features using the gain ratio attribute evaluation algorithm.

Feature	**Weight**	**Feature**	**Weight**
Return Type	1.0000	Out Calls	0.9678
Arguments	1.0000	#Out Calls	0.9678
Tracelets	0.9815	#Instructions	0.9644
Mnemonic Group	0.9815	#Operands	0.9644
Strings	0.9815	#Basic Blocks	0.9621
Constants	0.9815	*Arguments	0.9528
Imports	0.9815	#Arguments	0.9528
#Imports	0.9815	#Constants	0.9392
#Tracelets	0.9787	*Local Variables	0.9230
#Strings	0.9787	API Tags	0.9230
Calls	0.9785	Function Flags	0.1
#Calls	0.9785	#Library Tags	0.1
*Function	0.9691	#API Tags	0.1

Table 9: Feature Weights

The details of the fingerprint similarity score calculation are described in Algorithm 2. The Jaccard similarity coefficient is used to calculate the similarity of the global features and tracelet features of two fingerprints. These scores are then combined to form the final similarity score.

Algorithm 2: Computing Fingerprint Similarity Score

```
Input: A target fingerprint

Input: A candidate fingerprint

Output: Similarity score

Function Compute_Similarity(target_fingerprint, candidate_fingerprint)

similar_count = 0

tracelet_similarity = 0

target_features = target_fingerprint.global_features

candidate_features = candidate_fingerprint.global_features

/* Computing similarity of global features                          */

for feature in target_features do
    if feature.key in candidate_features then
        if target_features.feature == candidate_features.feature then
            similar_count += weight(feature.key)
        else
            if feature.isdigit() then
                if target_features.feature < candidate_features.feature then
                    similar_count += weight(feature.key) *
                     (target_features.feature / candidate_features.feature)
                else
                    similar_count += weight(feature.key) *
                     (candidate_features.feature / target_features.feature)
                end
            end
        end
    end
end

/* Continued on next page...                                        */
```

Algorithm2: Computing Fingerprint Similarity Score - Continued

```
/* Computing similarity of tracelets                                    */
target_tracelets = target_fingerprint.tracelets
candidate_tracelets = candidate_fingerprint.tracelets
similar_count += weight(number_of_tracelets) * min(len(target_tracelets),
 len(candidate_tracelets)) / max(len(target_tracelets),
 len(candidate_tracelets))
for tracelet in target_tracelets do
    tracelet_similarity = compare(tracelet, candidate_tracelets)
    similar_count += weight(tracelet) * tracelet_similarity
end
/* Computing final score using Jaccard index                           */
total_features = len(target_features) + len(target_tracelets) +
 len(candidate_features) + len(candidate_tracelets)
score = similar_count / (total_features - similar_count)
return score
end function
```

3.6 Summary

In this chapter, we have presented the detailed design and algorithms of our fingerprinting and matching approaches. We have described all the necessary steps for feature extraction and fingerprint generation in order to transform a binary function into a compact, meaningful fingerprint. The fingerprint is a combination of the function's extracted global features, tracelet features, and signature hash. The global

features are extracted once from the function as a whole. After decomposing the function's CFG into tracelets, the tracelet features are extracted from the basic blocks in each tracelet. The signature hash is generated through the min-hashing process of the function's normalized instructions.

After fingerprint generation, in order to match a target fingerprint against a large repository of binary functions, we first use the filtering process to reduce the dataset and acquire a candidate set of functions from the repository that are the most similar to the target fingerprint. Candidate functions pass through two different filters in order to be included in the candidate set. The first filter is concerned with the number of basic blocks in the function's CFG. The difference in the number of basic blocks of the target function and candidate function must be within 30% of the number of basic blocks of the target function. This filter is put in place because it is unlikely for a relatively small function to be highly similar in functionality to a significantly larger function. The second filter is based on the LSH banding technique of the signature hash, which represents the similarity of the functions' normalized instructions.

After reducing the functions in the dataset to a candidate set, a similarity score is computed between the target fingerprint and each fingerprint in the candidate set. The combination of the Jaccard similarity of the functions' global features and tracelet features represents the similarity score of the two functions. The functions are then ranked in a descending order based on the similarity score. In Chapter 4, we evaluate these fingerprinting and matching techniques through several experiments and state the results.

Chapter 4

Evaluation and Experimental Results

In this chapter, we perform several experiments in order to evaluate multiple aspects of BinSign's methodologies including its accuracy, performance, and scalability. These experiments comprise of function matching, function reuse detection, library function detection, malware similarity analysis, and function indexing scalability. We provide some insight into the effects of different compiler optimization levels and some obfuscation techniques on our fingerprint matching methodology. We also compare BinSign's results with the results of other tools in terms of function matching. Moreover, we describe BinSign's web-based user interface. In the following, we discuss the experiments and results in detail.

4.1 Dataset Description

We apply our experiments on a dataset that consists of widely used, well-known libraries and malware samples. Table 10 summarizes the details of the functions included in our dataset.

Function Type	File Name	#Functions	Size
Library Functions	libpng1.5.22-MSVC2010	525	250 KB
	libpng1.5.22-MSVC2013	539	257 KB
	libpng1.6.17-MSVC2010	604	277 KB
	libpng1.6.17-MSVC2013	620	285 KB
	sqlite3.8.1-MSVC2010	2,290	965 KB
	sqlite3.8.1-MSVC2013	2,308	1.006 MB
	sqlite3.8.9-MSVC2010	1,460	525 KB
	sqlite3.8.9-MSVC2013	1,471	565 KB
	zlib1.2.5-MSVC2010	164	92 KB
	zlib1.2.5-MSVC2013	169	93 KB
	zlib1.2.6-MSVC2010	179	92 KB
	zlib1.2.6-MSVC2013	183	93 KB
	zlib1.2.7-MSVC2010	174	92 KB
	zlib1.2.7-MSVC2013	178	93 KB
	zlib1.2.8-MSVC2010	174	92 KB
	zlib1.2.8-MSVC2013	178	93 KB
Malware Functions	Citadel	896	4.505 MB
	Zeus	642	2.705 MB
Noise Functions	Cassandra	5,493	1.39 MB
	SFTP	620	207 KB
	AdobeARM	4,702	1.01 MB
Total:	-	23,569	14.687 MB

Table 10: Dataset Details

The dataset includes different versions of the libraries: `libpng`, `sqlite`, and `zlib`. Version 1.5.22 and version 1.6.17 of the library `libpng` are used. Versions 3.8.1 and 3.8.9 are used of the library `sqlite`. Versions 1.2.5 through 1.2.8 of the `zlib` library are used. The dataset includes 16 different library files. These files are compiled using `Visual Studio` (`MSVC`) compilers 2010 and 2013. `Zeus` and `Citadel` botnet malware samples are also included in the dataset. The dataset includes 642 binary functions from `Zeus` and 896 binary functions from `Citadel`. We also include functions from the applications AdobeARM.exe, Cassandra.dll, and SFTP.exe as noise functions to increase the number of functions in the dataset. The total number of binary functions in the dataset is 23,569 functions. In order to evaluate the scalability of our system,

we add functions of system dynamic library files obtained from `Microsoft Windows` operating system to generate fingerprints for 6 million functions.

4.2 Comparison with Existing Tools

We perform this experiment in order to compare the accuracy of BinSign against `Diaphora` [3] and `PatchDiff2` [8] when used to match similar functions. The tools `Diaphora` and `PatchDiff2` are both IDA Pro plugins that can be used for comparing binary files. After comparing two binary files, the tools produce a mapping between the similar functions of the two files. In this experiment, we compare two versions of the same binary file. For each function in the target binary file, we use BinSign to match that target function with the corresponding function in the other binary file such that the matching function is the one with the highest similarity score on top of the candidate set. `Diaphora` offers different options when matching binary functions. We deactivate the option of using unreliable methods. We also activate the option of ignoring the function names since we assume that function names are not available during the matching process. We only use the function names as the ground truth for verification purposes.

To perform this experiment, we use the libraries `libpng`, `sqlite`, and `zlib` compiled using two different compilers. We then compare the two binary files of each library resulting from the compilation using the `MSVC` 2010 and `MSVC` 2013 compilers. By using two different compilers, we introduce some noise into the binary functions so there are some differences introduced by the compilers between the functions. After

that, we attempt to match the functions in the file compiled by MSVC 2010 as the target set against the functions in the file compiled by MSVC 2013. We match each function in the target set using BinSign by finding the function in the candidate set with the highest similarity score to that specific function. The function names in both binary files are stripped away during this experiment and are not used as part of the matching process. However, we verify the correct matches using the function names in the program debug database as the ground truth. If the candidate with the highest similarity score does not have the same name as the target function, we examine both functions manually in order to determine whether the function is the correct match or not.

Tool Name	Library	#Target Functions	#Correct Matches	Accuracy
Diaphora	libpng	620	408	65.81%
	sqlite	1489	657	44.12%
	zlib	156	79	50.64%
PatchDiff2	libpng	620	510	82.26%
	sqlite	1489	937	62.92%
	zlib	156	122	78.21%
BinSign	libpng	620	553	89.19%
	sqlite	1489	1391	93.42%
	zlib	156	134	85.90%

Table 11: Function Matching Comparison Between Tools

We display the results of the comparison in Table 11. The accuracy is calculated by finding the percentage of the correctly matched functions to the total number of functions in the binary file. Diaphora's accuracy when matching these library files ranges between 44.12% and 65.81%. The resulting accuracy of PatchDiff2 when matching these files ranges between 62.92% and 82.26%. BinSign achieves an accuracy

that ranges between 85.90% and 93.42% when matching these binary files. Table 11 shows that BinSign consistently achieves the highest accuracy between the tools being compared. This is due to the fuzziness of the method BinSign is using to perform the matching. This allows BinSign to be more lenient when dealing with the modifications presented by different compilers.

4.3 Function Reuse Detection

Through this experiment, we attempt to detect reused functions. The function reuse detection is performed between different versions of the same library and between different libraries. Different versions of the well-known `zlib` library are used for this experiment. The experiment is performed using versions 1.2.5, 1.2.6, 1.2.7, and 1.2.8 of this library. In order to detect reused functions between different libraries, we also use version 1.6.17 of the `libpng` library. Each fingerprint in the dataset must pass through two filters in order to be considered in the candidate set. Therefore, only functions with the number of basic blocks similar enough to the candidate function and with a similar min-hash signature to the one of the target function are included in the candidate set. We perform this experiment using different similarity thresholds for the hash signature to observe how this change affects the accuracy of the matching process. As mentioned previously in Section 3.5.1, an approximation of the threshold is calculated as $(\frac{1}{b})^{\frac{1}{r}}$, where b represents the number of bands and r represents the size of each band. Therefore, we set the threshold once at 60% (30 bands with 7 hash values in each band) and once at 65% (30 bands with 8 hash values in each band).

This experiment is performed on a PC with an Intel Core i7 CPU 920 @2.67 GHz and 12 GB of RAM running `Microsoft Windows` 7 64-bit.

4.3.1 Function Reuse Between `Zlib` Versions

In this experiment, we attempt to match the functions with the same symbolic name since we assume they perform similar functionality throughout the different versions of the library. Each version is used to attempt to match the corresponding functions in its consecutive version of the library. The function is considered to be matched correctly if the corresponding function in the other version of the library is ranked first with the highest similarity score among the set of candidate functions. If the candidate with the highest similarity score does not have the same name as the target function, we examine both functions manually in order to determine if the function is reused or not.

The results of this experiment are shown in Table 12. The accuracy is calculated by finding the percentage of the correctly matched functions to the total number of functions in the binary file. It is worth mentioning that although the accuracy is computed using the total number of functions in each binary file, not all functions are necessarily being reused in the consecutive version of the library. The number of candidates presented in Table 12 represents the sum of the sizes of the candidate sets acquired when matching all target functions. In other words, the number of candidates represents the number of functions with fingerprints that passed through the two filters: number of basic block filtering and LSH. The average time denotes the average time it takes for matching each function and acquiring a candidate set. As

we can see from the results, changing the LSH threshold from 60% to 65% decreases the accuracy score. This is expected since increasing the threshold can result in a true match being filtered out by the LSH filter.

Library Versions	#Target Functions	LSH Threshold	Accuracy	#Candidates	Average Time
zlib1.2.5 - zlib1.2.6	169	60% 65%	89.47% 78.11%	24970 11372	3.9s 2.8s
zlib1.2.6 - zlib1.2.7	183	60% 65%	94.67% 90.71%	30881 14947	3.7s 2.5s
zlib1.2.7 - zlib1.2.8	178	60% 65%	98.79% 88.76%	26965 12003	4.5s 3.7s
zlib1.2.8 - libpng1.6.17	52	60% 65%	100% 100%	4474 2147	4.7s 4.4s

Table 12: Results of Function Reuse Detection

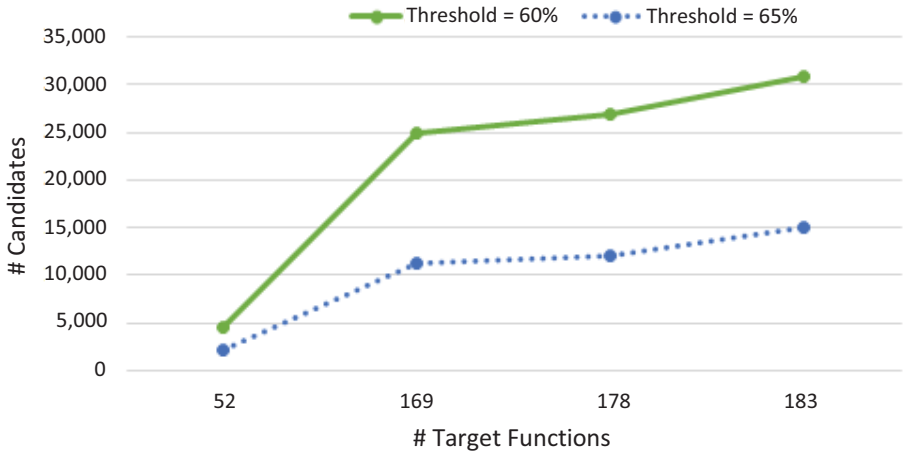


Figure 12: Number of Target Functions vs. Number of Candidates

Figure 12 plots the number of target functions when matching each of the library versions against the total number of candidates when the LSH threshold is set to 60% and 65%. We can see that increasing the number of functions being matched increases the total number of candidates as expected. However, there are other factors that affect the total number of candidates. The size of the target functions is one of these

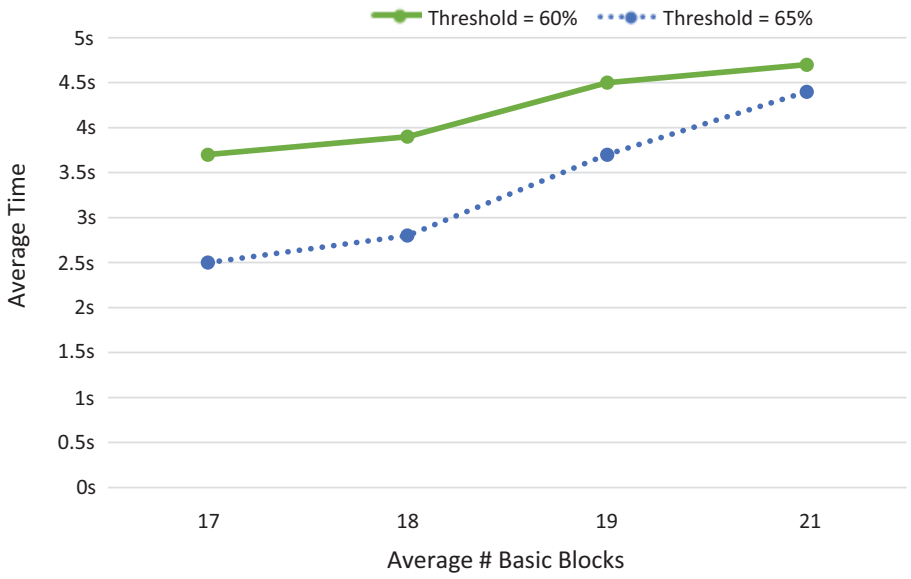


Figure 13: Number of Basic Blocks of Target Functions vs. Matching Time

factors. Smaller functions tend to contain fewer distinctive features. As a result, a smaller target function tends to have a larger candidate set, which increases the total number of candidates. Therefore, the total number of candidates does not only depend on the number of target functions, but also on the characteristics and size of the target functions. It is clear that the slope decreases in Figure 12. Therefore, we can deduce that a further increase in the number of target functions results in a smaller increase in the total number of candidates.

Figure 13 shows the effect of the number of basic blocks of the target functions on the matching time. Most of the time spent is due to computing the similarity scores between the target function and each candidate function. A smaller portion of that time is spent on obtaining the candidate set. Therefore, the number of candidates and the size of the functions are important factors that affect the matching time. The larger the functions being compared, the more time it requires for the matching

process to be completed. It is also clear that the slope decreases in Figure 13. Therefore, we can deduce that a further increase in the number of basic blocks of the target functions results in a smaller increase in the matching time.

4.3.2 Function Reuse Between Libraries

We also attempt to match reused functions from the `zlib` library in the `libpng` library as shown in the last row of Table 12. IDA Pro identifies 178 binary functions in version 1.2.8 of the `zlib` library and 620 binary functions in version 1.6.17 of the `libpng` library. We inspect the functions in both libraries manually and identify 52 reused functions. We use BinSign to attempt to match these functions from `zlib` as our target functions. BinSign is able to identify all 52 functions successfully with similarity scores ranging from 60% to 100%. In this case, changing the threshold does not affect the accuracy of the matching process since the correct matches still pass the filter even after modifying the size of the bands in the LSH filter.

4.4 Scalability Evaluation

In this experiment, we evaluate the scalability of the indexing process of fingerprints of binary functions into the `Cassandra` database. To this end, we index 6 million binary functions using BinSign and measure the time that the indexing process takes. These functions include library functions, malware samples, and functions of system dynamic library files obtained from `Microsoft Windows` operating system.

Before each function is indexed into the database, various features are extracted from the function and the function fingerprint is generated. The resulting fingerprint

is then stored in the database. On average, the indexing process requires around 0.0072 seconds per function. This includes the time necessary for fingerprint generation and communication with the database.

4.4.1 Fingerprint Methodology Scalability

Our fingerprinting methodology takes into consideration scalability when generating the fingerprint. This is accomplished through the min-hashing process. By using min-hashing and LSH, we enhance the scalability by selecting a candidate set through the banding technique as described in Section 3.5.1. Only candidates that are selected through our min-hashing and LSH filter and the number of basic blocks filter are considered instead of performing brute force matching in order to speed up the matching process. Therefore, the similarity score is calculated only between the candidate fingerprints that pass through these filters and the target fingerprint being matched.

4.4.2 Implementation Scalability

To improve the scalability of BinSign, we implement a distribution mechanism using a message queuing software, namely RabbitMQ [9]. The latter is an open source messaging software based on the international standard Advanced Message Queuing Protocol (AMQP) [1]. Thanks to its simplicity, we find that a messaging mechanism is more suitable to our purposes than other distribution frameworks that require a lot of processing/tools for data analysis and synchronization with the server, leading to a lot of overhead, and thus slowing the process.

Figure 14 depicts an overview of the distribution process using RabbitMQ. When

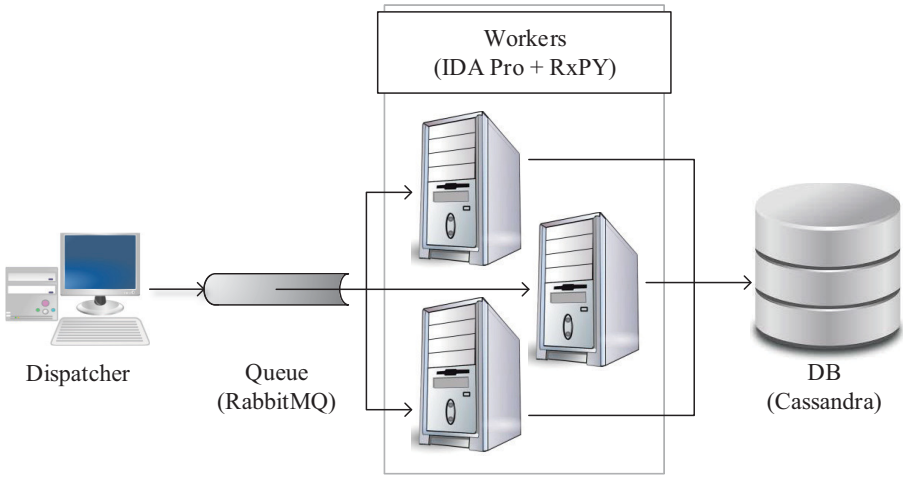


Figure 14: Architecture of the Distribution Process

indexing several binary files, the binaries are distributed to different workers. The distribution is done through a queue (Rabbit MQ) using Round-Robin scheduling. The queue sends binaries to each worker depending on the processing power of the worker machine. RabbitMQ uses a publish/subscribe pattern, so each worker subscribes to receive messages from the publisher. Each worker machine runs multiple instances of IDA Pro simultaneously in order to process the binary files, generate the functions' fingerprints, and store them in the `Cassandra` [2] database. Note that RxPY [10] (reactive extension) is used to run the code in an asynchronous manner. RxPY is a python library for composing asynchronous and event-based programs. In our experiments, the distribution is performed on a server machine with an Intel Xeon CPU E5-2630 v3 @2.40 GHz (2 processors) and 128 GB of RAM running `Microsoft Windows` Server 2008 64-bit, along with a PC with an Intel Core i7 CPU 920 @2.67 GHz and 12 GB of RAM running `Microsoft Windows` 7 64-bit. Additional worker machines can be used in order to further improve the scalability and performance of the system. In addition to RabbitMQ, the built-in pooling mechanism of

`Cassandra` [34] is also used to perform concurrent reads and writes to the database to improve performance and scalability.

4.5 Resilience to Different Compiler Optimization Levels

Through this experiment, we study the effects of using different compiler optimization levels on the accuracy of our fingerprint matching approach. We use `MSVC` 2013 to compile version 1.2.8 of the `zlib` library with four different optimization levels. `MSVC` 2013 offers the following optimization levels: disabled optimization, minimize size, maximize speed, or full optimization. These optimization levels are represented by the abbreviations: Od, O1, O2, and Ox, respectively. We compile the `zlib` library with full optimization (Ox) and use the resulting binary functions as the target functions in this experiment. We attempt to match the target functions with `zlib` functions compiled with the other optimization levels (Od, O1, and O2).

The results of this experiment are presented in Table 13. As expected, the lowest accuracy score of 65.05% occurs when matching the fully optimized (Ox) functions with the functions that were compiled with disabled optimization (Od). Compiling with optimization level O1 produces more similar functions to the fully optimized functions, which results in the higher accuracy score of 87.85%. We find that compiling with the optimization levels Ox and O2 seem to produce identical assembly code in this case. Therefore, the accuracy of the matching process is 100%.

After taking a closer look, we find that the size of the target functions being

Optimization Levels	Overall Accuracy	Average Time	#Basic Blocks	Accuracy
Ox vs. Od	65.05%	4.2s	5	62.5%
			10	66.7%
			15	53.8%
			20	60.0%
			25	100.0%
Ox vs. O1	87.85%	4.3s	5	83.3%
			10	83.3%
			15	84.6%
			20	100.0%
			25	100.0%
Ox vs. O2	100.00%	0.26s	5	100.0%
			10	100.0%
			15	100.0%
			20	100.0%
			25	100.0%

Table 13: Results of Matching Different Optimization Levels

matched has an effect on the accuracy of the matching process when matching different optimization levels. The accuracy of matching functions with different number of basic blocks is displayed in Figure 15. It is consistent among the different optimization levels being compared that the functions with higher number of basic blocks are matched with higher accuracy. This is due to the fact that larger functions tend to contain more distinctive features than smaller functions. Small functions usually have features and structures that are fairly common. Therefore, there is a higher probability of mismatching smaller functions than larger ones.

It is notable that the accuracy of matching between the optimization levels Ox and Od is consistently lower than matching between the optimization levels Ox and O1. This is due to the fact that when compiling with optimization level Od, none of the optimization techniques are applied to the code. This results in code with very

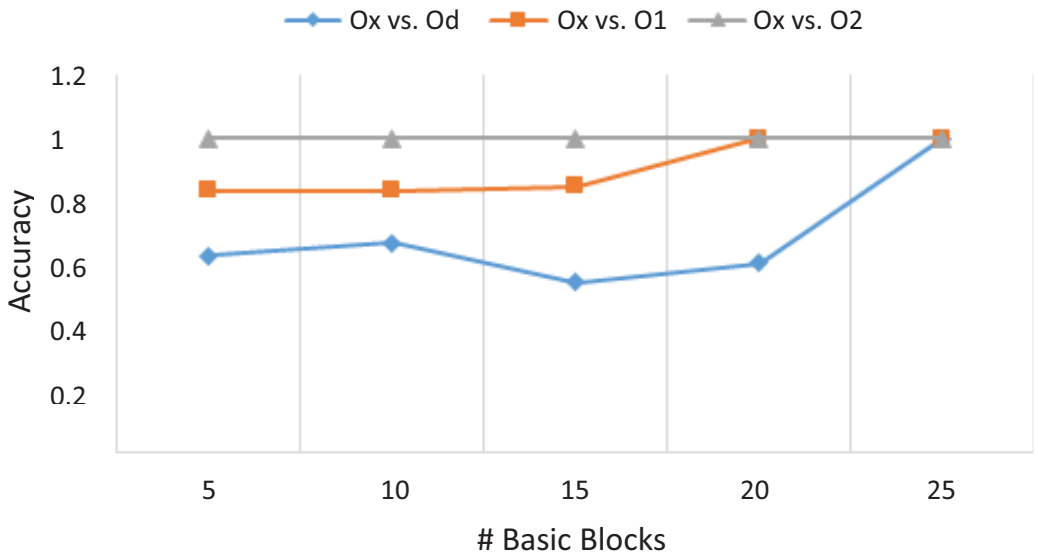


Figure 15: Matching Different Optimization Levels

different characteristics than code that has undergone some optimization (such as minimizing size or maximizing speed). However, optimization level O1 applies some optimization techniques that minimize the size, which results in code with higher similarity to the fully optimized functions. Compiling the code with the optimization level O2 seems to produce identical functions to the fully optimized code regardless of the size of the function.

4.6 Library Function Detection

Identifying library functions in binaries is useful in the reverse engineering process. The recognition of library functions assists the analysts in understanding the context of the binary piece of code in a more rapid and more accurate manner. Another benefit of identifying library functions is to decrease the number of binary functions that call for manual inspection by the reverse engineer. IDA F.L.I.R.T. is a technology

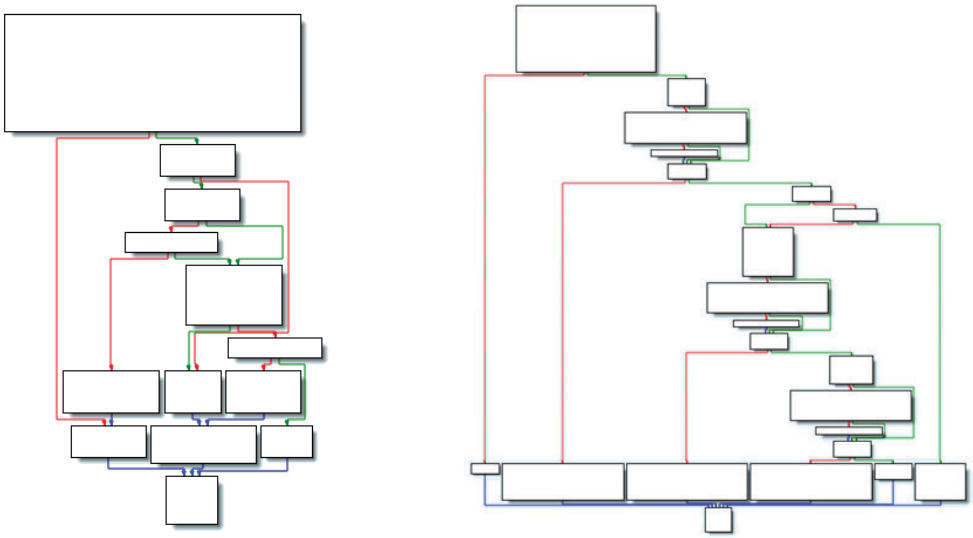
provided by the industry-standard disassembler IDA Pro. It provides a built-in capability for recognizing standard library functions by assigning labels to the identified standard library functions through exact matching of byte-level sequences. As a result, F.L.I.R.T. fails to detect the library function in the case of any slight byte-level discrepancies. In this experiment, we identify a few cases where IDA F.L.I.R.T. does not detect the library functions in the binary files and attempt to match these library functions using BinSign. We identify such library functions in two files: Putty.exe and Heap.exe. Table 14 and Table 15 show the results of our experiments on these executables respectively.

Function Name	**FLIRT Detection**	BinSign **Score**
_memchr	Yes	1.0
_memcpy	Yes	1.0
_memset	Yes	1.0
_memcpy_s	Yes	1.0
_memcmp	No	-

Table 14: Library Function Identification in Putty.exe

Table 14 shows that F.L.I.R.T. is able to detect the library functions except the last function, _memcmp. BinSign also exhibits similar results. It is worth noting that the similarity scores of the library functions matched by BinSign is 1.0. This is why F.L.I.R.T. is able to detect these functions through its exact matching technique. However, the version of the _memcmp function used in this binary is significantly different from the one used in F.L.I.R.T. to generate the target signature. The differences in the functions' CFGs can be seen in Figure 16. The differences in the basic blocks are significant enough such that BinSign is not able to match the function.

Table 15 shows the results of this experiment on the file Heap.exe. In this case,



(a) Original CFG

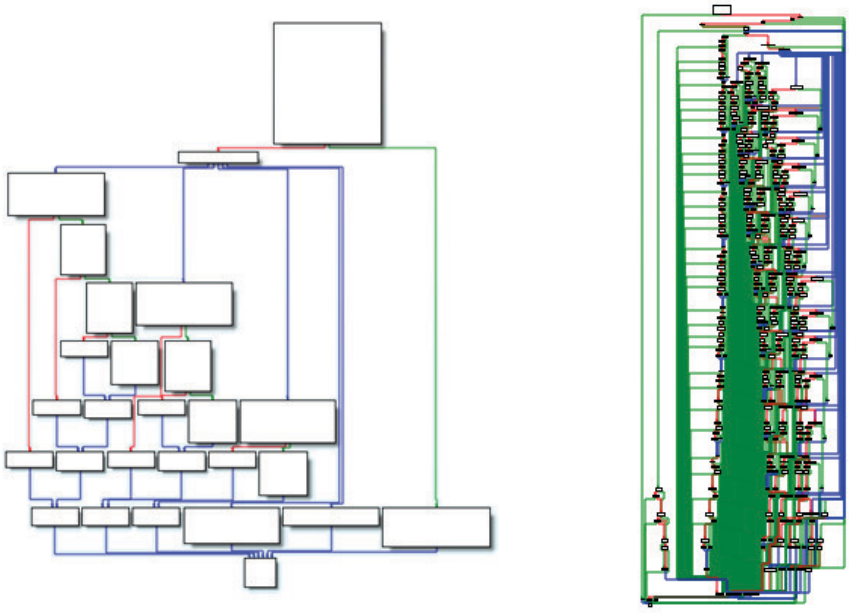
(b) Modified CFG

Figure 16: CFG of Different Versions of _memcpy_s

Function Name	FLIRT Detection	BinSign Score
_memchr	No	0.890
_memcpy	No	0.644
_memset	No	0.637
_memcpy_s	No	-
_memcmp	No	-

Table 15: Library Function Identification in Heap.exe

F.L.I.R.T. is not able to detect any of the library functions in this binary. However, BinSign is able to detect the first three library functions in the table with relatively high similarity scores. As for the functions _memcpy_s and _memcmp, neither F.L.I.R.T. nor BinSign were able to identify them. The differences in the basic blocks of the CFGs of these functions are significant as in the previous case. The variation in the CFG of the function _memcmp is even more drastic than _memcpy_s and the number of basic blocks increased radically as displayed in Figure 17.



(a) Original CFG

(b) Modified CFG

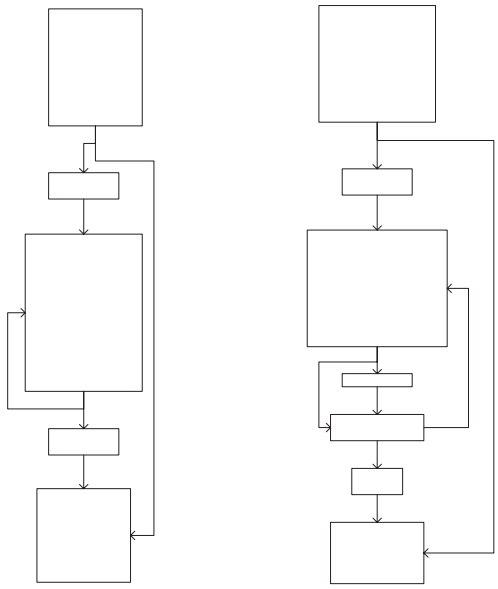
Figure 17: CFG of Different Versions of _memcmp

4.7 Malware Similarity Analysis

This experiment is performed using `Citadel` and `Zeus` malware samples. Since the functions in `Citadel` are derived from the functions in `Zeus` with some modifications [61], we identify the RC4 function in both malware samples and attempt to match the two versions of this function. `Citadel` reuses the RC4 stream cipher function from `Zeus` with minor modifications [69]. We attempt to identify the fingerprint of the RC4 function in `Zeus` using the fingerprint of the reused RC4 function from `Citadel`. We use IDA Pro to disassemble both `Zeus` and `Citadel`. IDA Pro identifies 642 functions in `Zeus` and 896 functions in `Citadel`. We then generate fingerprints of all the binary functions included in these malware samples and attempt to match the RC4 function. Functions from the previously mentioned library files are also included in the dataset. The time it takes to match the RC4 function is 0.463 seconds.

Function	Similarity Score
sub_42E92D	0.68787
sub_10034D0A	0.40042
png_set_sCAL	0.35377

Table 16: Candidates of RC4 Function from Citadel



(a) Zeus (b) Citadel

Figure 18: CFG of RC4 Function in Zeus and Citadel

After manual inspection, we find that the top match is in fact the modified RC4 function with a similarity score of 0.68787. The other matches with lower similarity scores are from different library files in the dataset. The CFG of both RC4 functions in Zeus and Citadel can be seen in Figure 18a and Figure 18b respectively. There are some clear modifications in the structure of the CFG of the function. Although the number of basic blocks in the two CFGs is different, the RC4 function is still identified because the difference in the number of basic blocks is not more than 30% of the number of basic blocks in the target function. Therefore, the matching RC4

function passes through the filters of BinSign and is selected as the function with the highest similarity score.

Since many of the functions in `Citadel` are derived from the functions in `Zeus`, we attempt to match all the functions in `Zeus`. BinSign matches 591 out of 642 functions in `Zeus` to functions in `Citadel`. Out of the matched functions, 546 functions are matched with a similarity score above 70%, 532 functions are matched with a similarity score above 80%, and 517 functions are matched with a very high similarity score above 90%.

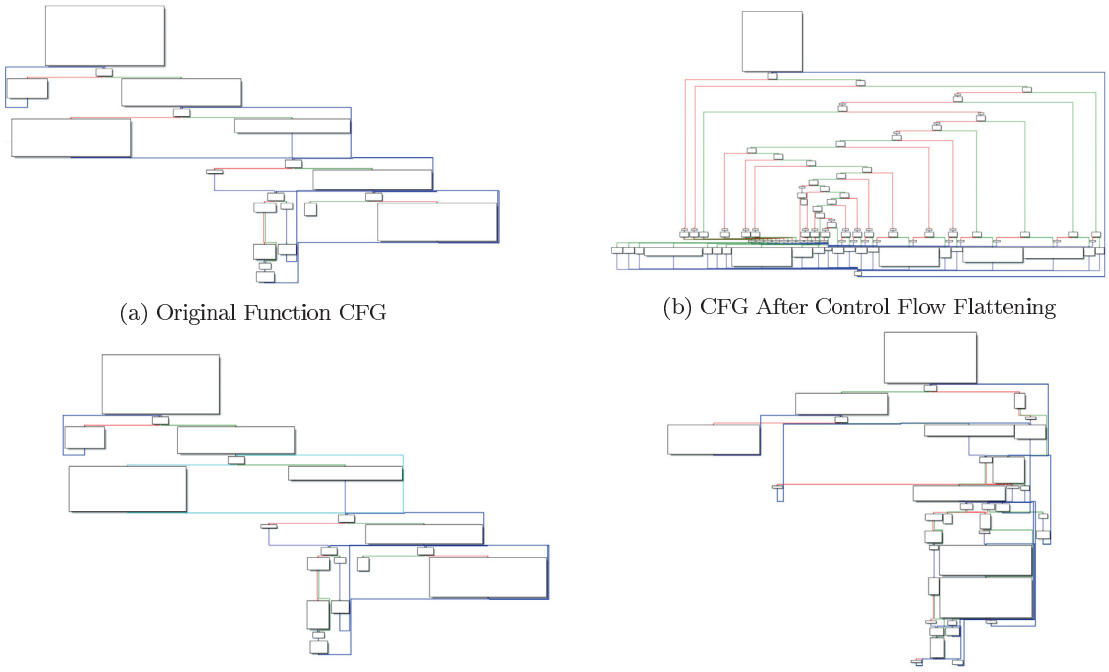
4.8 Resilience to Obfuscation

Different types of obfuscation techniques can be used that make the code more difficult to understand [21,66]. These techniques are used in order to prevent the reverse engineering of a software program. An obfuscator is a tool that automatically applies obfuscation techniques to alter a piece of code in such a way that obscures the code, making it more difficult to understand or reverse-engineer while preserving its functionality [11].

Although heavy obfuscation is out of the scope of our threat model, we provide some insight into the effects that some obfuscation techniques have on our fingerprint matching methodology. We use the Obfuscator-LLVM [7] for this purpose. It offers three different obfuscation techniques: control-flow flattening, instruction substitution, and bogus control-flow [50]. The purpose of the first technique is to fully flatten the CFG of a binary function. The control-flow flattening technique offered

by Obfuscator-LLVM is based on the technique described in more detail in [57]. The second obfuscation technique entails randomly selecting equivalent instruction sequences to replace standard binary operators in a manner that renders the code to be more complicated while maintaining the same functionality. The bogus control-flow technique [57] adjusts the call graph of a function by adding a new basic block before the current basic block and then makes a conditional jump to the original basic block. Moreover, the original basic block is cloned and populated with random, junk instructions.

After applying the three previously described obfuscation techniques to a piece of code written in C++ [63], we add the obfuscated functions to the dataset. We use the original function as the target function. The original function's CFG can be seen in Figure 19a. Consequently, the function that results from instruction substitution is identified as a match with a similarity score of 0.84399. The function is still identified with a high similarity score which shows resilience to this type of obfuscation. We illustrate in Figure 19c the CFG structure after applying the instruction substitution obfuscation technique. While this type of obfuscation does not modify the structure of the function's CFG, it does change the number and type of instructions inside each basic block. Figure 20 shows the instructions contained in two basic blocks as examples of the changes that occur after applying the instruction substitution obfuscation technique. There are visible changes in the number and types of instructions between the basic blocks before and after going through this type of obfuscation. The number of instructions significantly increased as shown in Figure 20b and Figure 20d.



(a) Original Function CFG

(b) CFG After Control Flow Flattening

(c) CFG After Instruction Substitution

(d) CFG After Bogus Control Flow

Figure 19: Effects of Obfuscation Techniques on a Function's CFG

However, the obfuscated functions resulting from control-flow flattening and bogus control-flow are not identified since the instructions and control flow structure are too different from the original function as displayed in Figure 19b and Figure 19d. The resulting CFGs are significantly different from the original function's CFG. The number of basic blocks increased by more than 30%. Therefore, the fingerprints of these functions are filtered out by our filtering mechanism which considers the number of basic blocks of the CFG.

```
lea    eax, _ZSt3cin@@GLIBCXX_3_4
lea    ecx, [ebp-18h]
mov    edx, [ebp-1Ch]
shl    edx, 2
add    ecx, edx
mov    [esp+58h+var_58], eax
mov    [esp+58h+var_54], ecx
call   __ZNSirsERi    ; std::istream::operator>>(int &)
mov    [ebp-38h], eax
mov    eax, [ebp-1Ch]
add    eax, 1
mov    [ebp-1Ch], eax
jmp    loc_80487F1
```

(a) First Basic Block Before Instruction Substitution

```
lea    eax, _ZSt4cout@@GLIBCXX_3_4
lea    ecx, aValueAt  ; "\t\t\tValue at "
mov    [esp+68h+var_68], eax
mov    [esp+68h+var_64], ecx
call   __ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
; std::operator<<<std::char_traits<char>>(std::basic_ostream<char,std::char_traits<char>> &,char const*)
mov    ecx, [ebp-24h]
mov    [esp+68h+var_68], eax
mov    [esp+68h+var_64], ecx
call   __ZNSolsEi     ; std::ostream::operator<<(int)
lea    ecx, aIndex    ; " Index: "
mov    [esp+68h+var_68], eax
mov    [esp+68h+var_64], ecx
call   __ZStlsISt11char_traitsIcEERSt13basic_ostreamIcT_ES5_PKc
; std::operator<<<std::char_traits<char>>(std::basic_ostream<char,std::char_traits<char>> &,char const*)
mov    ecx, [ebp-24h]
mov    ecx, [ebp+ecx*4-1Ch]
mov    [esp+68h+var_68], eax
mov    [esp+68h+var_64], ecx
call   __ZNSolsEi     ; std::ostream::operator<<(int)
lea    ecx, __ZSt4endlIcSt11char_traitsIcEERSt13basic_ostreamIT_T0_ES6_
; std::endl<char,std::char_traits<char>>(std::basic_ostream<char,std::char_traits<char>> &)
mov    [esp+68h+var_68], eax
mov    [esp+68h+var_64], ecx
call   __ZNSolsEPFRSoS_E ; std::ostream::operator<<(std::ostream & (*)(std::ostream &))
mov    [ebp-48h], eax
xor    eax, eax
mov    ecx, [ebp-24h]
mov    edx, eax
sub    edx, ecx
mov    ecx, eax
sub    ecx, 1
add    edx, ecx
sub    eax, edx
mov    [ebp-24h], eax
jmp    loc_8048882
```

(b) First Basic Block After Instruction Substitution

```
mov    eax, [ebp-2Ch]
mov    eax, [ebp+eax*4-18h]
mov    [ebp-24h], eax
mov    eax, [ebp-2Ch]
mov    eax, [ebp+eax*4-14h]
mov    ecx, [ebp-2Ch]
mov    [ebp+ecx*4-18h], eax
mov    eax, [ebp-24h]
mov    ecx, [ebp-2Ch]
mov    [ebp+ecx*4-14h], eax
```

(c) Second Basic Block Before Instruction Substitution

```
xor    eax, eax
mov    ecx, [ebp-30h]
mov    ecx, [ebp+ecx*4-1Ch]
mov    [ebp-28h], ecx
mov    ecx, [ebp-30h]
mov    edx, eax
sub    edx, 1
sub    ecx, edx
mov    ecx, [ebp+ecx*4-1Ch]
mov    edx, [ebp-30h]
mov    [ebp+edx*4-1Ch], ecx
mov    ecx, [ebp-28h]
mov    edx, [ebp-30h]
mov    esi, eax
sub    esi, edx
mov    edx, eax
sub    edx, 1
add    esi, edx
sub    eax, esi
mov    [ebp+eax*4-1Ch], ecx
```

(d) Second Basic Block After Instruction Substitution

Figure 20: Examples of Instruction Substitution in Basic Blocks

4.9 Web-Based User Interface

The binary function fingerprinting and detection tool we developed can run either inside the IDA Pro environment as an IDA Pro plugin or can be used through the web-based user interface. We build a simple, easy to use web interface in collaboration with a binary clone detection tool, BinSequence [47]. The interface enables the user to perform the following functionalities:

- Add a binary file to the target set.
- Add a binary file to the dataset.
- Select one or more target function(s) for fingerprint matching.
- Select one or more binary files from the dataset for fingerprint detection.
- View a list of functions with similar fingerprints to the target fingerprint(s) and compare the similarity scores.

The primary screen is depicted in Figure 21. It is divided into two panes. The pane on the left side shows the target set, and the pane on the right side shows the dataset. The list displayed below the label "Target Functions" in the left pane contains all the binary files and their functions that have been previously uploaded and can be used as target fingerprints during the fingerprint matching process. Initially, the list only displays the names of the binary files. Once the user clicks on a file name, the list expands to show all the function names in that file as shown in Figure 22. The numbers written inside the blue circles placed beside the name of each binary file indicate the total number of functions contained in that file and how many of

them are currently selected as target functions. The user can add a new binary file to this list by clicking on the blue button at the top right corner of the pane labeled as "Upload a binary". The user can also search for a specific file or function through the search bar at the top of the pane.

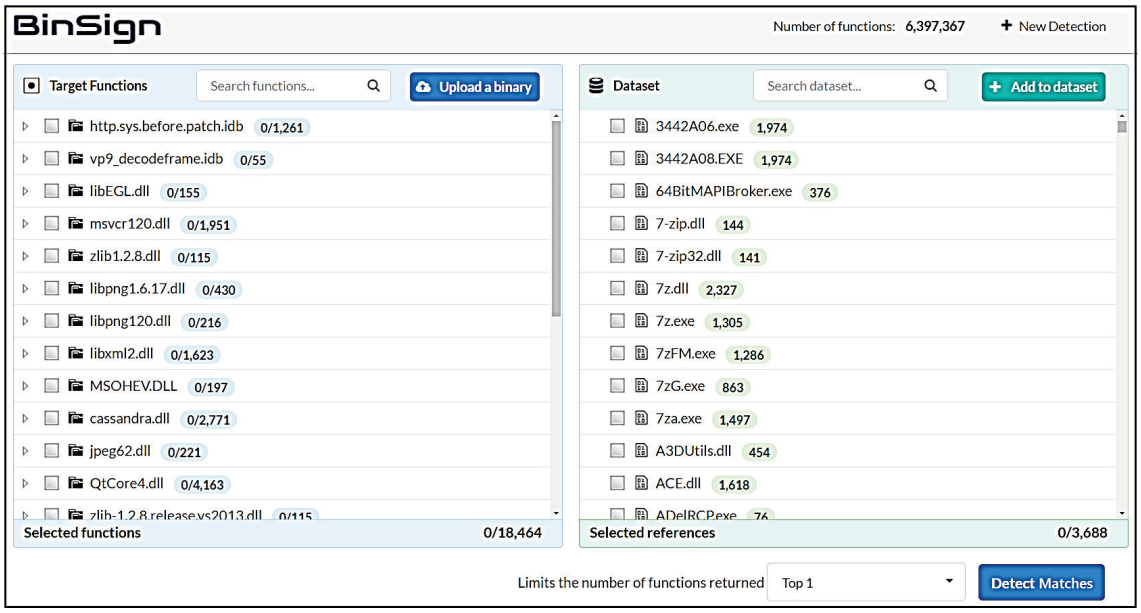


Figure 21: Primary Screen of Web-Based User Interface

The dataset pane on the right side in Figure 21 shows a list of binary files that have been previously uploaded to the dataset. The user can choose to match the target fingerprint(s) against the entire repository of functions, or the user can select one or more of these files so that the target fingerprint(s) are only matched against the selected binary files. The numbers written inside the green circles placed beside the name of each binary file indicate the total number of functions contained in that file. The user can add a new binary file to the dataset by clicking on the green button at the top right corner of the pane labeled as "Add to dataset". The user can also search for a specific file in the dataset through the search bar at the top of the pane.

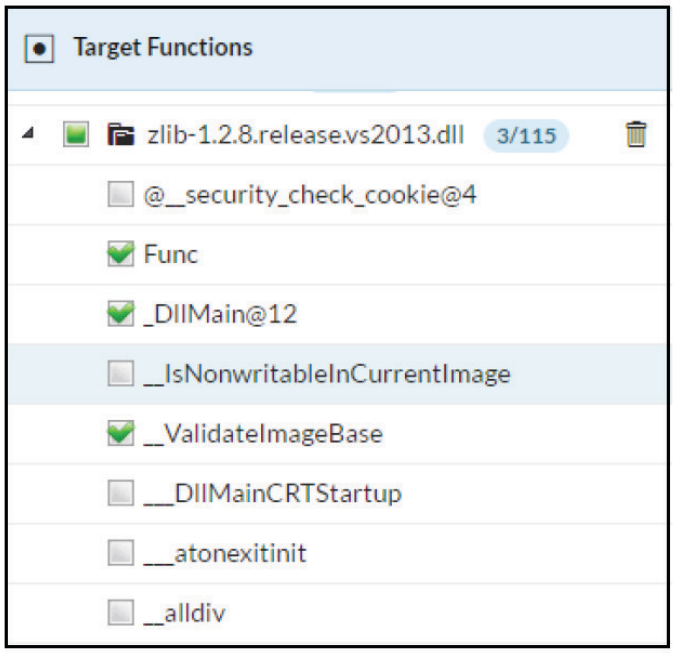


Figure 22: Interface of Selecting Target Fingerprint(s)

In order to find functions that are similar to a target fingerprint, the user selects one or more target function(s) from the target list by selecting the check box next to the function name as displayed in Figure 22. In this example, the functions "Func", "_DllMain@12", and "_ValidateImageBase" are selected as target functions from the file "zlib-1.2.8.release.vs2013.dll". The numbers next to the file name indicate that three functions are selected as target functions out of a total of 115 functions in this file. It is worth mentioning that the user can remove the file from the target list by clicking on the recycle bin icon next to the file name. The user then selects one or more file(s) from the dataset in order to match the target function(s) against by selecting the check box next to the file name as displayed in Figure 23. In this example, five files are selected. The user then clicks on the button labeled "Detect Matches" at the bottom right corner of the page.

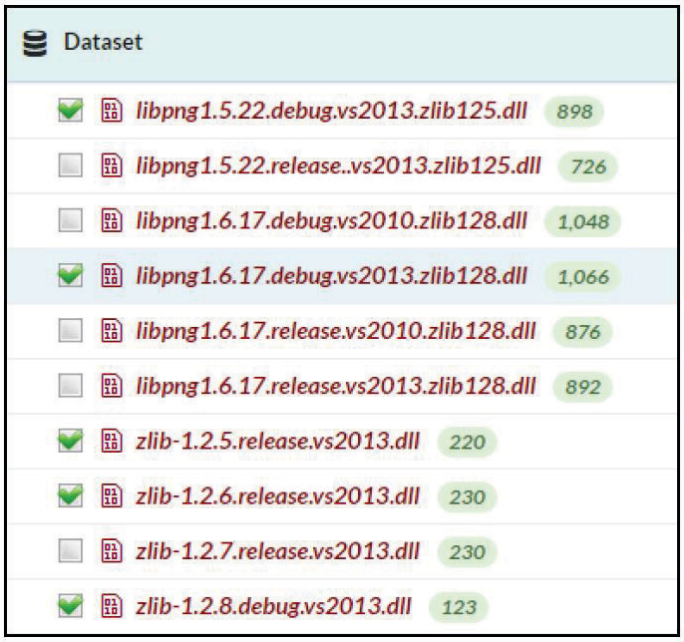
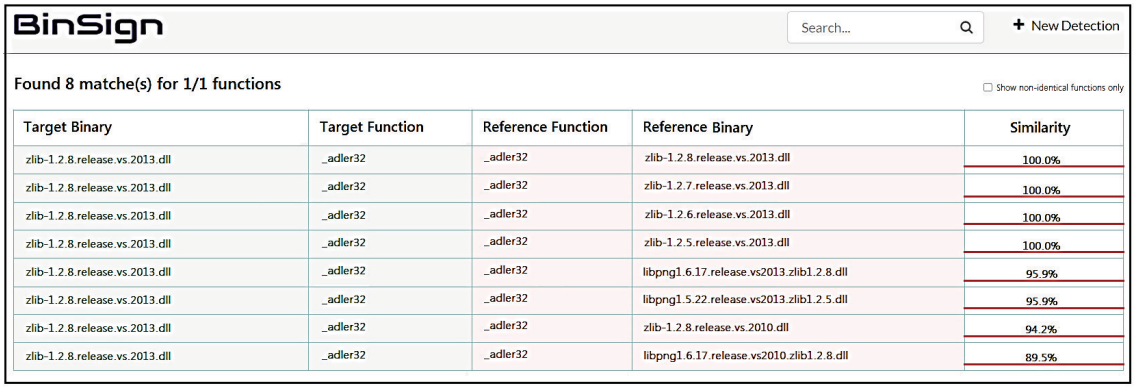


Figure 23: Interface of Selecting Dataset Files



Target Binary	Target Function	Reference Function	Reference Binary	Similarity
zlib-1.2.8.release.vs.2013.dll	_adler32	_adler32	zlib-1.2.8.release.vs.2013.dll	100.0%
zlib-1.2.8.release.vs.2013.dll	_adler32	_adler32	zlib-1.2.7.release.vs.2013.dll	100.0%
zlib-1.2.8.release.vs.2013.dll	_adler32	_adler32	zlib-1.2.6.release.vs.2013.dll	100.0%
zlib-1.2.8.release.vs.2013.dll	_adler32	_adler32	zlib-1.2.5.release.vs.2013.dll	100.0%
zlib-1.2.8.release.vs.2013.dll	_adler32	_adler32	libpng1.6.17.release.vs2013.zlib1.2.8.dll	95.9%
zlib-1.2.8.release.vs.2013.dll	_adler32	_adler32	libpng1.5.22.release.vs2013.zlib1.2.5.dll	95.9%
zlib-1.2.8.release.vs.2013.dll	_adler32	_adler32	zlib-1.2.8.release.vs.2010.dll	94.2%
zlib-1.2.8.release.vs.2013.dll	_adler32	_adler32	libpng1.6.17.release.vs2010.zlib1.2.8.dll	89.5%

Figure 24: Interface of Displaying Fingerprint Matching Results

The result of the matching process is then displayed as in Figure 24. The screen contains a table summarizing the results of the matching process. The table contains the name of the target binary file, the name of the target function(s), the name of the reference function(s) from the dataset, and the name of the binary file from the dataset that contains each reference function. The last column of the table displays

the similarity scores between the target fingerprint and each reference fingerprint from the dataset. The functions are ranked in descending order based on the similarity scores such that the function with the highest similarity score is displayed at the top row of the table.

4.10 Summary

In this chapter, we have presented several experiments conducted in order to evaluate multiple aspects of BinSign's methodologies as well as the details of the web-based user interface of BinSign. We have evaluated BinSign in terms of its accuracy, performance, and scalability. Our dataset is comprised of several well-known libraries, malware samples, and functions of system dynamic library files obtained from `Microsoft Windows` operating system. Our experiments show that BinSign outperforms existing tools and achieves a high accuracy score. We also described different measures undertaken to ensure BinSign's scalability. BinSign performs efficient fingerprint generation such that the indexing process requires an average time of 0.0072 seconds per function. Our experiments also show that BinSign's matching process is resilient to light obfuscation techniques (light instruction substitution).

Chapter 5

Conclusion and Future Work

Disassemblies consist of a number of functions with unknown functionality, which require manual analysis by a reverse engineer in order to gather useful information about their functionalities. A line-by-line review of assembly instructions is a tedious and time-consuming task. It requires a working knowledge of low-level system-dependent instructions and their context. Automated binary analysis frameworks are required to support the engineer during the reverse engineering process. The objective of these frameworks is to classify binary functions according to their functionality and context, and filter out known functions such as recognized library functions.

In this thesis, we reviewed different approaches to function fingerprinting. The reviewed function fingerprint generation approaches include hashing techniques and extracting syntactic, semantic, and/or graph-based features for the purpose of fingerprint generation. We also reviewed different fingerprint matching techniques including exact and inexact matching schemes. The reviewed literature used function fingerprinting for several purposes including library identification, clone detection,

search-based fingerprinting, and malware classification.

Moreover, we presented an efficient, accurate, and scalable binary function fingerprinting tool named BinSign. We defined the main components of function fingerprinting, described the proposed algorithms, and the details of the binary fingerprint generation and matching processes. Our fingerprint generation methodology combines different syntactic, semantic, and structural features from the function's CFG in the fingerprint generation process. The structure of the function's CFG is captured through decomposing the CFG into partial execution traces called "tracelets". Min-hashing is also used in order to generate the signature hash which allows for a scalable and efficient fingerprint matching process. To improve the scalability of the fingerprint matching process, we designed and implemented two filters to decrease the number of functions under consideration from the dataset and acquire a smaller, more refined candidate set when matching a target function fingerprint. The functions in the dataset must pass through the filtering of the number of basic blocks and the signature hash filtering in order to be included in the candidate set. The Jacquard similarity index is used on two different levels (function global features and tracelet features) to calculate the similarity score between the target fingerprint and each fingerprint in the candidate set. The functions are then ranked in descending order based on their similarity scores.

We evaluated BinSign's accuracy, performance, and scalability in several experiments. The proposed methodologies were evaluated in terms of function matching, function reuse detection, malware similarity analysis, library function detection, resilience to different compiler optimization levels, resilience to obfuscation techniques,

and scalability. We used binary functions from well-known libraries and malware samples during our experiments. Through these experiments, we showed that the proposed methodologies are effective in facilitating the analysis of binary functions and improving the accuracy of exact and inexact binary fingerprint matching. We also demonstrated the scalability of BinSign by indexing 6 million function fingerprints. The indexing process required an average time of 0.0072 seconds per function.

The experiments also uncovered some limitations in our approach which can be addressed in possible future research works. The limitations mainly centralize around the issues of heavy code obfuscation. We briefly describe these limitations and suggest some solutions as possible future research directions.

- Function Inlining: Compilers may use function inlining for code optimization. Some overhead results from calling and returning from a function. This overhead can be eliminated by eliminating the function call and expanding the body of the caller function to include the code of the called function. If a target function is the function that is inlined within another function, then our approach may not be able to match it. However, function inlining is usually performed on relatively small functions. The functionality of these small functions is generally straightforward and can easily be analyzed by reverse engineers. Furthermore, if a target function is modified to encompass another small function, then there is a high probability for our approach to be able to successfully match these fingerprints due to the fuzziness of our matching process.

- Control-Flow Flattening: Control-flow flattening is one of the techniques that

can be used to obfuscate binary code and make it more difficult to understand and reverse engineer. This obfuscation technique involves modifying the code structure in a manner that makes it more difficult to identify the targets of branches through static analysis, which hinders the understanding of the code [57,57]. When comparing fingerprints of binary functions, we take into consideration the CFG structure, which is captured through the tracelet features. Therefore, transforming the CFG structure using control-flow flattening could obstruct the matching process. Future research work may take into account cases where merging basic blocks by removing unneeded jumps such as unconditional branching instructions can result in a more effective matching process.

- Bogus Control-Flow: Bogus control-flow is another obfuscation technique that can be used to complicate the understanding of binary code. This technique entails modifying the CFG of a function by adding a conditional jump with two outward edges pointing either to the original basic block or to a fake added basic block with junk instructions that loops back to the conditional jump block [57]. Since this technique also modifies the structure of the CFG, it could pose a problem during the fingerprint matching process. Future research work may involve identifying unreachable and dead code in order to disregard this code during the fingerprint generation process.

Extending BinSign in future research work to consider these points may result in fingerprint generation and matching processes that are more resilient to code obfuscation techniques.

Bibliography

[1] Advanced Message Queuing Protocol (AMQP). Available at: https://www.amqp.org/. Last accessed: January 2017.

[2] Apache Cassandra Web site. Available at: http://cassandra.apache.org/. Last accessed: January 2017.

[3] Diaphora: A Program Diffing Plugin for IDA Pro. Available at: https://github.com/joxeankoret/diaphora. Last accessed: January 2017.

[4] Hex-Rays IDA Pro. Available at: https://www.hex-rays.com/products/ida/. Last accessed: January 2017.

[5] Hungarian Algorithm. Available at: http://www.hungarianalgorithm.com/. Last accessed: January 2017.

[6] Internet Security Threat Report 2016. Available at: https://www.symantec.com/content/dam/symantec/docs/reports/istr-21-2016-en.pdf. Last accessed: January 2017.

[7] Obfuscator-LLVM. Available at: https://github.com/obfuscator-llvm/obfuscator/wiki. Last accessed: January 2017.

[8] PatchDiff2: Binary Diffing Plugin for IDA. Available at: https://code.google.com/p/patchdiff2/. Last accessed: January 2017.

[9] RabbitMQ Web site. Available at: https://www.rabbitmq.com/. Last accessed: January 2017.

[10] The Reactive Extensions for Python. Available at: https://github.com/ReactiveX/RxPY. Last accessed: January 2017.

[11] Thicket Family of Source Code Obfuscators. Available at: http://www.semdesigns.com/Products/Obfuscators/. Last accessed: January 2017.

[12] Weka: Machine Learning Software. Available at: https://weka.wikispaces.com/. Last accessed: January 2017.

[13] Alrabaee, S., Saleem, N., Preda, S., Wang, L., and Debbabi, M. Oba2: an onion approach to binary code authorship attribution. *Digital Investigation 11* (2014), S94–S103.

[14] Alrabaee, S., Shirani, P., Debbabi, M., and Wang, L. On the feasibility of malware authorship attribution. In *International Symposium on Foundations and Practice of Security* (2016), Springer, Cham, pp. 256–272.

[15] Alrabaee, S., Shirani, P., Wang, L., and Debbabi, M. Sigma: A semantic integrated graph matching approach for identifying reused functions in binary code. *Digital Investigation 12* (2015), S61–S71.

[16] Alrabaee, S., Wang, L., and Debbabi, M. Bingold: Towards robust binary analysis by extracting the semantics of binary code as semantic flow graphs (sfgs). *Digital Investigation 18* (November 2016), S11–S22.

[17] Andoni, A., and Indyk, P. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)* (2006), IEEE, pp. 459–468.

[18] Andriesse, D., Chen, X., van der Veen, V., Slowinska, A., and Bos, H. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *USENIX Security Symposium* (2016).

[19] Bania, P. Generic unpacking of self-modifying, aggressive, packed binary programs. *arXiv preprint arXiv:0905.4581* (2009).

[20] Bayer, U., Kirda, E., and Kruegel, C. Improving the efficiency of dynamic malware analysis. In *Proceedings of the 2010 ACM Symposium on Applied Computing* (March 2010), ACM, pp. 1871–1878.

[21] Behera, C. K., and Bhaskari, D. L. Different obfuscation techniques for code protection. *Procedia Computer Science 70* (2015), 757–763.

[22] Bonfante, G., Marion, J. Y., Sabatier, F., and Thierry, A. Code synchronization by morphological analysis. In *Malicious and Unwanted Software (MALWARE), 2012 7th International Conference on* (October 2012), IEEE, pp. 112–119.

[23] Bourquin, M., King, A., and Robbins, E. Binslayer: accurate comparison of binary executables. In *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop* (2013), ACM, p. 4.

[24] Bruschi, D., Martignoni, L., and Monga, M. Detecting self-mutating malware using control-flow graph matching. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment* (July 2006), Springer Berlin Heidelberg, pp. 129–143.

[25] Cesare, S., and Xiang, Y. Classification of malware using structured control flow. In *Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing-Volume 107* (January 2010), Australian Computer Society, Inc., pp. 61–70.

[26] Cesare, S., and Xiang, Y. Taxonomy of program features. In *Software Similarity and Classification*. Springer London, 2012, pp. 7–16.

[27] Cesare, S., Xiang, Y., and Zhou, W. Control flow-based malware variant detection. *IEEE Transactions on Dependable and Secure Computing 11*, 4 (2014), 307–317.

[28] Chaki, S., Cohen, C., and Gurfinkel, A. Supervised learning for provenance-similarity of binaries. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining* (August 2011), ACM, pp. 15–23.

[29] Chandramohan, M., Xue, Y., Xu, Z., Liu, Y., Cho, C. Y., and Tan, H. B. K. Bingo: cross-architecture cross-os binary search. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (2016), ACM, pp. 678–689.

[30] Cifuentes, C., and Gough, K. J. Decompilation of binary programs. *Software: Practice and Experience 25*, 7 (1995), 811–829.

[31] Comparetti, P. M., Salvaneschi, G., Kirda, E., Kolbitsch, C., Kruegel, C., and Zanero, S. Identifying dormant functionality in malware programs. In *2010 IEEE Symposium on Security and Privacy* (May 2010), IEEE, pp. 61–76.

[32] Coogan, K., Debray, S., Kaochar, T., and Townsend, G. Automatic static unpacking of malware binaries. In *2009 16th Working Conference on Reverse Engineering* (October 2009), IEEE, pp. 167–176.

[33] Cordy, J. R., and Roy, C. K. Debcheck: Efficient checking for open source code clones in software systems. In *Program Comprehension (ICPC), 2011 IEEE 19th International Conference on* (June 2011), IEEE, pp. 217–218.

[34] DataStax. Connection Pooling. Available at: `http://docs.datastax.com/en/developer/java-driver/2.1/manual/pooling/`. Last accessed: January 2017.

[35] David, Y., and Yahav, E. Tracelet-based code search in executables. *ACM SIGPLAN Notices 49*, 6 (2014), 349–360.

[36] Ding, S. H., Fung, B. C., and Charland, P. Kam1n0: Mapreduce-based assembly clone search for reverse engineering. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, NY, USA, 2016), KDD '16, ACM, pp. 461–470.

[37] Dullien, T., and Rolles, R. Graph-based comparison of executable objects (english version). *SSTIC 5* (2005), 1–3.

[38] Egele, M., Scholte, T., Kirda, E., and Kruegel, C. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR) 44*, 2 (2012), 6.

[39] Elva, R., and Leavens, G. T. Semantic clone detection using method ioe-behavior. In *Proceedings of the 6th International Workshop on Software Clones* (June 2012), IEEE Press, pp. 80–81.

[40] Farhadi, M. R., Fung, B. C., Charland, P., and Debbabi, M. Binclone: Detecting code clones in malware. In *Software Security and Reliability (SERE), 2014 Eighth International Conference on* (June 2014), IEEE, pp. 78–87.

[41] Gascon, H., Yamaguchi, F., Arp, D., and Rieck, K. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM workshop on Artificial intelligence and security* (November 2013), ACM, pp. 45–54.

[42] Guilfanov, I. IDA fast library identification and recognition technology (FLIRT Technology): In-depth. Available at: `https://www.hex-rays.com/`

products/ida/tech/flirt/in_depth.shtml, 2012.

[43] Gupta, P., Singh, J., Arora, A. K., and Mahajan, S. Digital forensics: A technological revolution in forensic sciences. *Journal of Indian Academy of Forensic Medicine* (2011), 166–170.

[44] Harrold, M. J., Rothermel, G., and Orso, A. Representation and analysis of software. Available at: http://www.ics.uci.edu/~lopes/teaching/inf212W12/readings/rep-analysis-soft.pdf. Last accessed: January 2017.

[45] Hido, S., and Kashima, H. A linear-time graph kernel. In *2009 Ninth IEEE International Conference on Data Mining* (December 2009), IEEE, pp. 179–188.

[46] Hu, X., Chiueh, T. C., and Shin, K. G. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM conference on Computer and communications security* (November 2009), ACM, pp. 611–620.

[47] Huang, H. Binary code reuse detection for reverse engineering and malware analysis. Master's thesis, Concordia University, December 2015.

[48] Jacobson, E. R., Rosenblum, N., and Miller, B. P. Labeling library functions in stripped binaries. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools* (September 2011), ACM, pp. 1–8.

[49] Jin, W., Chaki, S., Cohen, C., Gurfinkel, A., Havrilla, J., Hines, C., and Narasimhan, P. Binary function clustering using semantic hashes. In *Machine Learning and Applications (ICMLA), 2012 11th International Conference*

on (December 2012), vol. 1, IEEE, pp. 386–391.

[50] Junod, P., Rinaldini, J., Wehrli, J., and Michielin, J. Obfuscator-llvm: software protection for the masses. In *Proceedings of the 1st International Workshop on Software Protection* (May 2015), IEEE Press, pp. 3–9.

[51] Karbab, E. B., Debbabi, M., and Mouheb, D. Fingerprinting android packaging: Generating dnas for malware detection. *Digital Investigation 18* (2016), S33–S45.

[52] Karegowda, A. G., Manjunath, A., and Jayaram, M. Comparative study of attribute selection using gain ratio and correlation based feature selection. *International Journal of Information Technology and Knowledge Management 2*, 2 (2010), 271–277.

[53] Keivanloo, I., Rilling, J., and Charland, P. Internet-scale real-time code clone search via multi-level indexing. In *2011 18th Working Conference on Reverse Engineering* (October 2011), IEEE, pp. 23–27.

[54] Khoo, W. M., Mycroft, A., and Anderson, R. Rendezvous: a search engine for binary code. In *Proceedings of the 10th Working Conference on Mining Software Repositories* (May 2013), IEEE Press, pp. 329–338.

[55] Kinable, J., and Kostakis, O. Malware classification based on call graph clustering. *Journal in computer virology 7*, 4 (2011), 233–245.

[56] Lakhotia, A., Preda, M. D., and Giacobazzi, R. Fast location of similar code fragments using semantic 'juice'. In *Proceedings of the 2nd ACM SIGPLAN*

Program Protection and Reverse Engineering Workshop (January 2013), ACM, p. 5.

[57] László, T., and Kiss, Á. Obfuscating c++ programs via control flow flattening. *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica 30* (2009), 3–19.

[58] Leskovec, J., Rajaraman, A., and Ullman, J. D. *Mining of massive datasets*. Cambridge University Press, 2014.

[59] Marcus, A., and Maletic, J. I. Identification of high-level concept clones in source code. In *Automated Software Engineering, 2001.(ASE 2001). Proceedings. 16th Annual International Conference on* (November 2001), IEEE, pp. 107–114.

[60] Mateljan, V., Juričić, V., and Peter, K. Analysis of programming code similarity by using intermediate language. In *MIPRO, 2011 Proceedings of the 34th International Convention* (May 2011), IEEE, pp. 1235–1240.

[61] Milletary, J. Citadel trojan malware analysis. *Luettavissa: http://botnetlegalnotice.com/citadel/files/Patel_Decl_Ex20.pdf.Luettu 13* (2012), 2014.

[62] Moser, A., Kruegel, C., and Kirda, E. Exploring multiple execution paths for malware analysis. In *2007 IEEE Symposium on Security and Privacy (SP'07)* (May 2007), IEEE, pp. 231–245.

[63] Munir, F. C Programming Tutorial For Beginners. Available at: http://fahad-cprogramming.blogspot.ca/2014/05/bubble-sort-in-c-code-example.html. Last accessed: January 2017.

[64] Ng, B. H., and Prakash, A. Expose: Discovering potential binary code re-use. In *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual* (July 2013), IEEE, pp. 492–501.

[65] Ohno, A., and Murao, H. Measuring source code similarity using reference vectors. In *First International Conference on Innovative Computing, Information and Control-Volume I (ICICIC'06)* (August 2006), vol. 2, IEEE, pp. 92–95.

[66] Popa, M. Techniques of program code obfuscation for secure software. *Journal of Mobile, Embedded and Distributed Systems 3*, 4 (2011), 205–219.

[67] Rahimian, A., Charland, P., Preda, S., and Debbabi, M. Resource: a framework for online matching of assembly with open source code. In *International Symposium on Foundations and Practice of Security* (October 2012), Springer Berlin Heidelberg, pp. 211–226.

[68] Rahimian, A., Shirani, P., Alrbaee, S., Wang, L., and Debbabi, M. Bincomp: A stratified approach to compiler provenance attribution. *Digital Investigation 14* (2015), S146–S155.

[69] Rahimian, A., Ziarati, R., Preda, S., and Debbabi, M. On the reverse engineering of the citadel botnet. In *Foundations and Practice of Security*. Springer International Publishing, 2014, pp. 408–425.

[70] Rosenblum, N. E., Miller, B. P., and Zhu, X. Extracting compiler provenance from program binaries. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering* (June 2010), ACM, pp. 21–28.

[71] Saidi, H., Yegneswaran, V., and Porras, P. Experiences in malware binary deobfuscation. *Virus Bulletin* (2010).

[72] Singh, R. A review of reverse engineering theories and tools. *International Journal of Engineering Science Invention 2*, 1 (2013), 35–38.

[73] Suarez-Tangil, G., Tapiador, J. E., Peris-Lopez, P., and Ribagorda, A. Evolution, detection and analysis of malware for smart devices. *IEEE Communications Surveys & Tutorials 16*, 2 (2014), 961–987.

[74] Tamboli, M. S., and Prasad, R. S. Authorship analysis and identification techniques: A review. *International Journal of Computer Applications 77*, 16 (2013).

[75] Treude, C., Figueira Filho, F., Storey, M. A., and Salois, M. An exploratory study of software reverse engineering in a security context. In *2011 18th Working Conference on Reverse Engineering* (October 2011), IEEE, pp. 184–188.

[76] Udupa, S. K., Debray, S. K., and Madou, M. Deobfuscation: Reverse engineering obfuscated code. In *12th Working Conference on Reverse Engineering (WCRE'05)* (November 2005), IEEE, pp. 10–pp.

[77] Van Emmerik, M. Identifying library functions in executable file using patterns. In *Software Engineering Conference, 1998. Proceedings. 1998 Australian* (November 1998), IEEE, pp. 90–97.

[78] Yianilos, P. N. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms* (Philadelphia, PA, USA, January 1993), SODA '93, Society for Industrial and Applied Mathematics, pp. 311–321.

[79] Yusirwan, S., Prayudi, Y., and Riadi, I. Implementation of malware analysis using static and dynamic analysis method. *International Journal of Computer Applications 117*, 6 (2015).