BINARY CODE REUSE DETECTION FOR REVERSE ENGINEERING AND MALWARE ANALYSIS

He Huang

A thesis

in

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

December 2015

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By:	**He Huang**
Entitled:	**BINARY CODE REUSE DETECTION FOR REVERSE ENGINEERING AND MALWARE ANALYSIS**

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. Jia Yuan Yu	______________________	Chair
Dr. Lingyu Wang	______________________	Examiner
Dr. Joey Paquet	______________________	Examiner
Dr. Mourad Debbabi	______________________	Supervisor
Dr. Amr Youssef	______________________	Supervisor

Approved ______________________________

Chair of Department or Graduate Program Director

______________ 20 ______ ______________________

Dr. Amir Asif, Dean

Faculty of Engineering and Computer Science

Abstract

BINARY CODE REUSE DETECTION FOR REVERSE ENGINEERING AND MALWARE ANALYSIS

He Huang

Code reuse detection is a key technique in reverse engineering. However, existing source code similarity comparison techniques are not applicable to binary code. Moreover, compilers have made this problem even more difficult due to the fact that different assembly code and control flow structures can be generated by the compilers even when implementing the same functionality. To address this problem, we present a fuzzy matching approach to compare two functions. We first obtain our initial mapping between basic blocks by leveraging the concept of longest common subsequence on the basic block level and execution path level. Then, we extend the achieved mapping using neighborhood exploration. To make our approach applicable to large data sets, we designed an effective filtering process using Minhashing and locality-sensitive hashing.

Based on the approach proposed in this thesis, we implemented a tool named BinSequence. We conducted extensive experiments to test BinSequence in terms of performance, accuracy, and scalability. Our results suggest that, given a large assembly code repository with millions of functions, BinSequence is efficient and can attain high quality similarity ranking of assembly functions with an accuracy above 90% within seconds.

We also present several practical use cases including patch analysis, malware analysis, and bug search. In the use case for patch analysis, we utilized BinSequence to compare the unpatched and patched versions of the same binary, to reveal the vulnerability information and the details of the patch. For this use case, a Windows system driver (`HTTP.sys`) which contains a recently published critical vulnerability is used. For the malware analysis use case, we utilized BinSequence to identify reused components or already analyzed parts in malware so that the human analyst can focus on those new functionality to save time and effort. In this use case, two infamous malware, Zeus and Citadel, are analyzed. Finally, in the bug search use case, we utilized BinSequence to identify vulnerable functions in software caused by copying and pasting or sharing buggy source code. In this case, we succeeded in using BinSequence to identify a bug from Firefox. Together, these use cases demonstrate that our tool is both efficient and effective when applied to real-world scenarios.

We also compared BinSequence with three state of the art tools: Diaphora, PatchDiff2 and BinDiff. Experiment results show that BinSequence can achieve the best accuracy when compared with these tools.

Acknowledgments

I would like to express my deepest gratitude to my advisors, Drs. Amr Youssef and Mourad Debbabi, for their enlightening guidance and tremendous support during my studies. Without their profound knowledge and precious insights, this thesis would not have been possible. Also, I greatly appreciate their dedication to helping students, not only academically, but also personally.

My gratefulness extends to the members of the examining committee: Drs. Lingyu Wang and Joey Paquet, who honored me by accepting to evaluate this thesis. Their time and efforts are highly appreciated.

My sincere thanks goes to my friends and lab-mates at Concordia University and Computer Security Lab, especially Gaby, Saed and Paria. I am grateful for the chance to be a member of this lab.

Last but not the least, special thanks to my family for their support and unconditional love.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Reverse engineering [33] is a primary step towards understanding the functionality and behavior of a software when its source code is not available. However, reverse engineering is a tedious and time-consuming process, and its success depends heavily on the experience and knowledge of the reverse engineer. Moreover, as the software to be analyzed grows in size, this task becomes overwhelming. Code reuse detection is thus of great interest to reverse engineers. For example, given a binary and a repository of already analyzed and commented code, one can speed up the analysis by applying code reuse detection on the binary to identify identical or similar code in the repository, and then focus only on the new functionality or components of the binary.

Consider, for instance, malware reverse engineering. Malware authors do not create viruses from scratch; instead, they tend to reuse their existing source code. Besides, in order to not reinvent the wheel, they may leverage some open source projects that provide certain functionality that they require. Identifying these reused code not only greatly reduces the efforts of analysis, but also helps in understanding the behavior of malware. For example, Citadel, derived from the leaked Zeus source code, keeps most of the core components of Zeus intact [62], and the malware Flame makes heavy use of SQLite [22], which is a light-weight database engine.

Code reuse detection is also of high interest to software maintainers and consumers. In many software development environments, it is common practice to copy and paste existing source code, as this can significantly reduce programming effort and time. However, if the copied code contains a bug or vulnerability, and the developers copied the code without fixing the bug, they may bring the bug into their own project. Library reuse is a special case in which the developer either includes the source code of a certain library into their project, or statically links to the library directly. Either way, the bug contained in the copied code will be brought into the new project. Code reuse detection can help identify such bugs resulting from shared source code.

Last but not least, code reuse detection can be applied in numerous scenarios such as software plagiarism detection, open source project license violation detection and binary diffing.

Code reuse detection can be achieved by calculating the similarity of two code regions. The higher the similarity, the more likely they are from the same source code base. In this thesis, we present an approach for measuring the similarity of two assembly functions.

1.2 Contributions

The contributions of our thesis can be summarized as follows:

- We designed a fuzzy matching approach to compare assembly functions. To address the mutations introduced by the compilers, our fuzzy matching algorithm operates at multiple levels, namely instruction level, basic block level and structure level.

- To prune the search space when comparing a target function against a vast number of functions, we designed an effective filtering process with two filters which can efficiently rule out functions that are not likely to be matched to our target function. With the help of this filtering process, we can compare one function against millions of functions within seconds.
- Based on the approach presented in this thesis, we implemented a fully working tool for binary code reuse detection. Extensive experiments show that our tool is fast, accurate, and scalable.
- We introduced many use cases, including patch analysis, malware analysis, and bug search, to demonstrate the efficiency and effectiveness of our approach when applied in real-world scenarios.

1.3 Thesis Organization

The remainder of the thesis is organized as follows. Chapter 2 provides a literature review and background knowledge of source code and binary code reuse detection. In Chapter 3, we introduce the fuzzy matching approach we use to compare functions and the detailed design of our filtering process. In Chapter 4, we present the results of the extensive experiments we conducted to evaluate our approach. Chapter 5 concludes the thesis and provides possible future research directions.

Chapter 2

Related Work and Background Knowledge

In general, when comparing two software for code reuse detection, there are three different scenarios according to the presence of the source code and binary code.

1. The two software are both available in source code form.

2. Only binary code is available.

3. One software is in source code form, while the other is in binary format.

For the first case, we can perform source code reuse detection. In the second scenario, binary code reuse detection can be applied. For the last scenario, binary to source matching, or source to binary matching, can be conducted.

In this chapter, we first introduce the state of the art techniques for source code reuse detection and binary code reuse detection. We classify these techniques into several categories, and for each category we introduce and briefly describe some of the most representative techniques.

2.1 Source Code Reuse Detection

Source code reuse detection, sometimes known as source code clone detection, is a well-explored area, and many approaches have been proposed in different literatures. In general, these approaches can be classified into five categories [70] [71], namely text-based approaches, token-based approaches, tree-based approaches, graph-based approaches and metrics-based approaches.

Text-based approaches

Text-based approaches consider the source code as a sequence of lines and compare the raw source code directly. Normally, prior to the actual comparison, little or no transformation/normalization is performed, except for basic steps such as comment removal and whitespace removal.

The pioneer paper by Johnson [42] presented a substring matching approach where the source files are first transformed to remove characters such as white spaces. A sliding window is then used to generate a set of substrings with a minimum length of 50 lines. When matching substrings, Johnson leverages a hash scheme based on the Karp and Rabin string matching approach [45, 46] to generate a fingerprint for each substring. The intention is to save storage space and matching overhead. To allow fuzzy matching, Johnson also normalizes the source code by replacing each maximal sequence of alphanumeric characters by a single letter, such as ‘x’.

For example, the line

$$for(k = 1; k <= n; k + +)$$

would be normalized to

$$x(x = x; x <= x; x + +)$$

and the line

$$\#define \quad ASDF \quad 1234$$

would be normalized to

$$\#xxx$$

Despite the loss of information by normalizing the code, the number of matches would not explode due to the requirement of a 50-line match.

Ducasse *et al.* [30] developed a language-independent visual approach to identify source code reuse. First, the source code is transformed to remove all comments and white spaces. Then, Ducasse *et al.* treat every line as an entity and compare it with every other line (entity) using string matching. The result is a comparison matrix. Subsequently, as shown in Figure 1, the scatter-plots are used to visualize the matrix. In scatter-plots, diagonals of dots represent lines of reused source code. To capture duplicated code that was changed inside one line, a pattern matcher is used to find diagonals with holes up to a certain size.

Marcus and Maletic [58] applied latent semantic indexing to detect high-level concept clones such as abstract data types. They use a simpler equivalency definition to reduce the cost and difficulty of detection at the expense of some lack in precision and automation. Furthermore, their method would fail to identify two functions with similar structure and functionality if comments are not available and the identifier names are different.

Token-based approaches

Token-based approaches normally first perform lexical analysis on the given source code. Subsequently, a sequence of tokens is extracted from the source. The sequence is then scanned for duplicated subsequences, and the corresponding source code is reported as reused code. Compared to text-based approaches, token-based approaches are generally more resilient to minor code changes such as formatting or spacing.

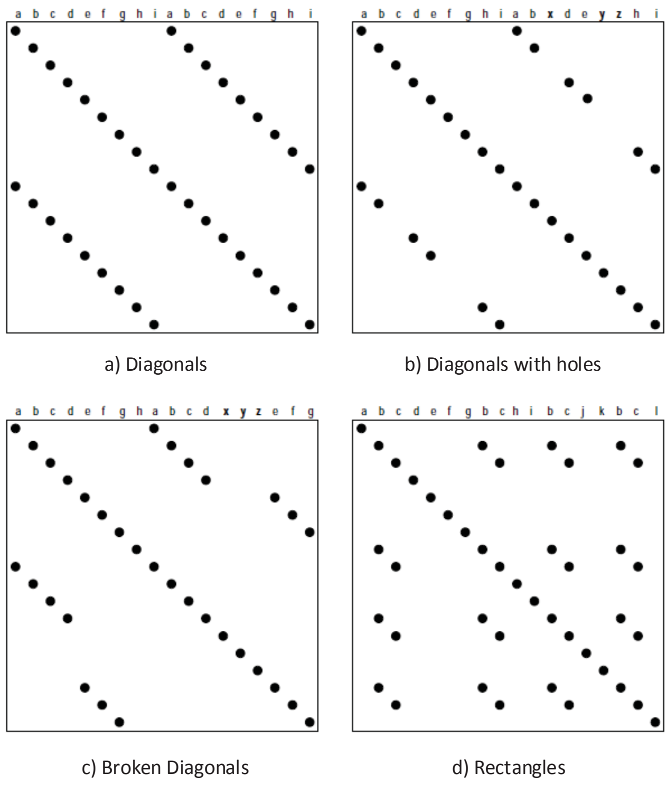

a b c d e f g h i a b c d e f g h i
a) Diagonals
a b c d e f g h i a b x d e y z h i
b) Diagonals with holes
a b c d e f g h a b c d x y z e f g
c) Broken Diagonals
a b c d e f g b c h i b c j k b c l
d) Rectangles

Figure 1: An example of different configurations of dots and each letter represents a line (Ducasse *et al.* [30])

Kamiya *et al.* [43] proposed a language-independent token-based source code clone detection tool named CCFinder. By transforming the input source code into a regular form of token sequence, CCFinder can extract code clones in multiple programming languages including C, C++, JAVA, and COBOL.

The first step of the approach introduced by Kamiya *et al.* is to perform lexical analysis on the given source files. During this step, each line is processed and divided into tokens based on the specific lexical rule of the programming language. All of the tokens from each source file are concatenated to form one single token sequence, and all of the white spaces and comments are removed. The generated token sequence is then transformed based on certain language-dependent transformation rules. For example, for C++ code, Kamiya *et al.* perform the following modifications: removing namespace attribution, removing template parameters, removing initialization lists, separating function definitions, removing accessibility keywords, and converting the source code into compound blocks (statements enclosed by braces).

Following these transformation rules,

```
void print_table (const map<string, string>& m) {
       int c = 0 ;
       map<string, string>::const_iterator i
              = m.begin();
       for (; i != m.end(); ++i) {
              cout << c << ", "
                     << i->first << " "
                     << i->second << endl;
              ++ c;
       }
}
```

will be transformed into the following token sequence:

```
void print_table ( const map & m ) {
int c = 0 ;
const_iterator i
= m . begin ( ) ;
for ( ; i != m . end ( ) ; ++ i ) {
cout << c << ", "
<< i -> first << " "
<< i -> second << endl ;
++ c ;
}
}
```

The next step is to replace all identifiers related to types, variables, and constants with a special token such as $\$p$. In so doing, code regions with similar structures but different syntax, like variable names, can be matched as clone pairs. For example, after replacing all identifiers, the above token sequence will become the following:

```
$p $p ( $p $p & $p ) {
$p $p = $p ;
$p $p
= $p . $p ( ) ;
for ( ; $p != $p . $p ( ) ; ++ $p ) {
$p << $p << $p
<< $p -> $p << $p
<< $p -> $p << $p ;
++ $p ;
}
}
```

All equivalent subsequence pairs of the transformed token sequence are now detected as clone pairs. In order to efficiently compute the matching, a suffix-tree matching algorithm [38] is adopted, such that the clone location information is represented as a tree with sharing nodes for leading identical subsequences. By searching leading nodes on the tree, all clone pairs can be identified.

Basit *et al.* [20] proposed an efficient token-based clone detection tool named *Repeated Tokens Finder (RTF).* Unlike most works that use suffix trees for string matching, RTF uses more efficient suffix arrays to detect string matches. Moreover, it provides a simple and customizable tokenization mechanism.

Basit *et al.* first use a language-specific tokenizer to assign each token class (e.g., keywords, operators, and comment markers) a unique numeric ID. Each token and its location are stored for output generation. In this step, all blank lines and comments are ignored and only one single large token string is generated from all source files. During tokenization, RTF gives the user multiple options to tailor the generated token string. First, RTF allows the user to suppress insignificant token classes. For example, access modifiers such as *private*, *protected*, and *public* do not carry much information in terms of clone detection and thus, this "noise" should be suppressed. RTF also allows the user to equate different token classes. For example, if users do not want to differentiate between the types {*int, short, long, float, double*} , they can use the same ID to represent every member of these types. The motivation is to match codes that only differ in the type of certain variables. In addition, the tokenizer they use can locate method boundaries to exclude clones that start from the middle of one method and end in the middle of another. By doing this, human analysts can focus on other meaningful clones.

Basit *et al.* treat clone detection as the problem of finding repeating substrings within the token string. Specifically, they focus only on finding non-extendible (NE) repeating substrings, as these NE repeating substrings correspond to clone classes

[43]. To this end, they leverage the suffix array data structure [57]. The key benefit of using the suffix array instead of a suffix tree is a significant reduction in memory usage. Subsequently, in order to locate all NE repeating pairs of substrings, a variation of the algorithm described in [16] is used so that the complete sets of NE repeats are computed instead of just pairs.

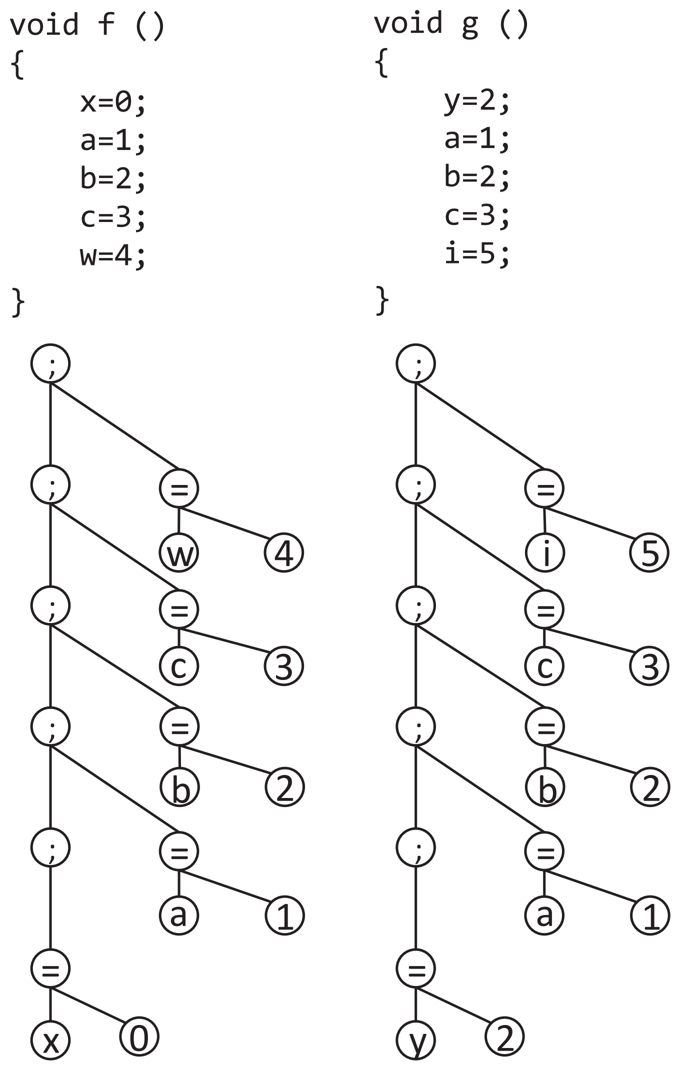
Tree-based approaches

Tree-based approaches normally first transform the source code into tree representations, such as parse trees or abstract syntax trees (ASTs). In these tree representations, variable names and literal values are abstracted away and similar subtrees represent code clones.

Yang [78] proposed one of the first approaches for identifying the syntactic differences from the source code of two versions of the same program. First, both the target and the reference source code are transformed into two parse trees using a parser. In this tree representation, a node denotes a token (e.g., variable name) or a non-terminal that represents a substructure (e.g., expression). The longest common subsequence algorithm is then applied, to match nodes of both parse trees. A node of a tree that does not have a matching node in the other tree is considered as a difference. However, in this approach, the source code must be syntactically correct. If part of the source code does not conform to the grammar, the parser will fail to produce a tree representation; consequently, the whole comparison will fail.

Baxter *et al.* [21] proposed a tool using ASTs to detect duplicated source codes, or in other words, clones. The first step of their approach is to parse the source code, and from this, produce an AST. Figure 2 shows an example of the generated AST. Subsequently, Baxter *et al.* apply three different algorithms on the generated ASTs to find clones.

Their first algorithm, called Basic algorithm, is used to detect subtree clones. Instead of comparing trees for exact equality, Baxter *et al.* choose to compare trees



```
void f ()
{
    x=0;
    a=1;
    b=2;
    c=3;
    w=4;
}
```

```
void g ()
{
    y=2;
    a=1;
    b=2;
    c=3;
    i=5;
}
```

Figure 2: Abstract syntax tree (Baxter *et al.* [21])

for similarity. To this end, Baxter *et al.* adopt the hashing technique which was used for building directed acyclic graphs for expressions in compiler construction [21]. First, all subtrees are hashed to several buckets, and only subtrees in the same bucket are compared for equality. This saves a great deal of computation at the expense of only a little additional storage space. Sometimes two subtrees are almost identical, except for a few differences. However, a good hash function would hash them into two different buckets. Baxter *et al.* denote such clones as near-miss clones. To detect near-miss clones, an artificially bad hash function is chosen so that such subtrees are hashed to the same bucket. Finally, the similarity between two subtrees is calculated using the following formula:

$$similairty = 2 \times S/(2 \times S + L + R)$$

where S is the number of shared nodes, L is the number of different nodes of the target subtree, and R is the number of different nodes in the reference subtree. If the calculated similarity is above a specified threshold, these two subtrees are added to the clone list.

The second algorithm is the sequence detection algorithm. Based on the Basic algorithm, the sequence detection algorithm focuses on right- or left-leaning trees with some kind of identical sequencing operator as a root. To detect these clone sequences, Baxter *et al.* leverage the longest common sequence algorithm to compare each pair of subtrees that contain sequence nodes, to look for maximum length sequences that encompassed previously detected clones.

After these two algorithms are applied, Baxter *et al.* begin to visit the parents of the already-detected clones and to verify whether their parents are near-miss clones. By doing this, more complex near-miss clones can be detected. In this step, subsumed clones are also deleted.

Graph-based approaches

The program dependency graph (PDG) is the most commonly used graph representation in source code reuse detection. In PDGs, nodes represent program statements and predicates, and edges represent data and control dependencies. Since the PDG is a high level abstraction of the original source code, approaches based on PDGs are normally resilient to multiple changes such as statement reordering or insertion, as long as such changes do not alter the original dependencies.

Komondoor and Horwitz [49] designed a PDG-based approach to identify source code clones. The first step of their approach is to represent each procedure using its program dependence graph. All PDG nodes are then partitioned into equivalence classes based on the syntactic structure of the statement/predicate that the node represents. For each pair of matching nodes (r_1, r_2), Komondoor and Horwitz find two isomorphic subgraphs of the PDGs that contain r_1 and r_2. To this end, they use two kinds of program slicing [77]: backward slicing and forwarding slicing. Take backward slicing for example, Komondoor and Horwitz start from r_1 and r_2 and slice backwards in lock step. A predecessor and the connecting edge are added to one slice if and only if there is a corresponding matching predecessor in the other PDG. The output of the slicing is two isomorphic subgraphs that represent duplicated source code. After all isomorphic subgraphs have been identified, Komondoor and Horwitz conduct post-processing steps, including removing subsumed clones and combining clones into larger groups. A clone pair (S_1', S_2') subsumes another clone pair (S_1, S_2) iff $S_1 \subseteq S_1'$ and $S_2 \subseteq S_2'$. So, given two clone pairs (S_1', S_2') and (S_1, S_2), (S_1, S_2) will be removed if (S_1, S_2) is subsumed by (S_1', S_2'). When combining clones into larger groups, two clone pairs (S_1, S_2) and (S_1, S_3) will be combined into one large clone group (S_1, S_2, S_3).

Liu *et al.* [55] developed a PDG-based software plagiarism detection tool named GPLAG. They state that even if the source code has been significantly altered, the corresponding PDG is nearly invariant. Liu *et al.* thus address the plagiarism detection problem through isomorphism testing. To tolerate some extent of noise, instead of doing exact isomorphism testing, they conduct γ-isomorphism testing by introducing a relaxation parameter γ ($0 < \gamma \leq 1$). A graph G is γ-isomorphic to G' if there exists a subgraph $S \subseteq G$ such that S is subgraph isomorphic to G', and $|S| \geq \gamma |G|, \gamma \in (0, 1]$. Intuitively, the γ-isomorphism testing becomes exact isomorphism testing when γ equals to 1.

Liu *et al.* also analyzed five types of alteration techniques that may be applied during plagiarism: format alteration, identifier renaming, statement reordering, control replacement, and code insertion. Format alteration involves adding/removing separators, blanks, or comments into/from the original source code. However, as the PDG is a high level abstraction of the code, this alteration does not change the PDG. Identifier renaming is to rename variables, classes, or procedures. Consequently, the syntax of the corresponding nodes of the original PDG may change, but the structures remain preserved. Statement reordering is a technique of changing the order of statements. However, there are normally some dependencies between statements. Some statements may perform operations on the output of previous statements. As a result, the order of these statements cannot be switched. On the other hand, reordering two instructions that are not bounded by dependencies will leave the PDG untouched. Then Liu *et al.* studied the effect of control replacement on the PDGs as well, and found that most replacements do not change the PDGs. The only exception is when a *while* or *for* loop is replaced by an infinite loop with a *break* statement. In this case, a vertex will be added to the new PDG. However, this added vertex does not break any existing dependencies. As a result, the original PDG is still an isomorphic subgraph of the new PDG. The last type of alteration, code insertion, normally introduces new vertices or edges into the new PDG. However, the old PDG is still an isomorphic subgraph of the new PDG, as the introduced vertices or dependencies do not change the existing vertices or dependencies in the original PDG.

Relaxed subgraph isomorphism testing is a time-consuming process. In order to prune the search space, Liu *et al.* adopt two filters: a lossless filter and a lossy filter. The lossless filter has two stages. In the first stage, PDGs with a size smaller than a threshold K will not be checked for isomorphism, as Liu *et al.* focus only on finding non-trivial PDG isomorphism pairs. The second stage is based on the definition of γ-isomorphism. Two PDGs g and g' will not be checked if $|g'| < \gamma|g|$, as they cannot be γ-isomorphic. The second filter, a lossy filter, is based on the vertex histogram of the PDG. Specifically, a tuple is used to represent each PDG:

$$h(g) = (n_1, n_2, ..., n_k)$$

where n_i is the frequency of the ith kind of vertex. Two PDGs g and g' are checked for γ-isomorphic only if the similarity between $h(g)$ and $h(g')$ is above a threshold. Experiment results suggest that, with the help of these filters, more than 90% of the original search space can be pruned.

Krinke [51] proposed a fine-grained PDG-based approach for identifying similar code. In Krinke's approach, the traditional PDGs are first transformed into fine-grained PDGs. To this end, vertices are attributed with a class, an operator, and a value. The class is used to specify the type of vertex, such as statement, expression, and procedure call. The operator further specifies the type, for example, binary expression or constant. The value is used to carry the exact operator, constant values, and identifier names. Krinke also separates the edges into three specialized types: immediate control edges, value dependence edges, and reference dependence edges. Immediate control edges denote the control dependence between the components of an expression, such that the targets of this type of edge are always evaluated prior to the source. Value dependence edges represent the data flow between the expression components. Reference dependence edges are used to denote the assignment of a computed value to a variable. After all vertices and edges in both PDGs are attributed, Krinke reports the maximal isomorphic subgraphs as duplicated code.

Metrics-based approaches

Metrics-based approaches first extract a number of metrics/vectors either from the source code directly or from other abstractions of the original code, such as abstract syntax trees. Instead of comparing source code, these metrics/vectors are compared. These approaches are based on the assumption that if two code fragments are similar, then the extracted metrics/vectors should be similar as well.

Kontogiannis *et al.* [50] proposed two techniques for detecting source code clones. The first technique extracts five well-known metrics from the source code and compares the metric values directly. The second technique leverages a dynamic programming algorithm to find the best alignment between two code fragments. Kontogiannis *et al.* use the following five metrics:

- The number of functions called (fanout)
- The ratio of input/output variables to the fanout
- McCabe cyclomatic complexity [60]
- Modified Albrecht's function point metric [17]
- Modified Henry-Kafura's information flow quality metric [39]

Assuming s is a code fragment, the McCabe cyclomatic complexity can be calculated as $McCabe(s) = 1 + d$, where d is the number of control decision predicates in s, and the modified Albrecht's function point metric can be calculated as:

$$\begin{aligned} Albrecht(s) = & P_1 \times VARS\_USED\_AND\_SET(s) + P_2 \times GLOBAL\_VARS\_SER(s) + \\ & P_3 \times USER\_INPUT(s) + P_4 \times FILE\_INPUT(s) \end{aligned}$$

where P_1, P_2, P_3, and P_4 are four weight factors. The modified Henry-Kafura's information flow quality metric is defined as:

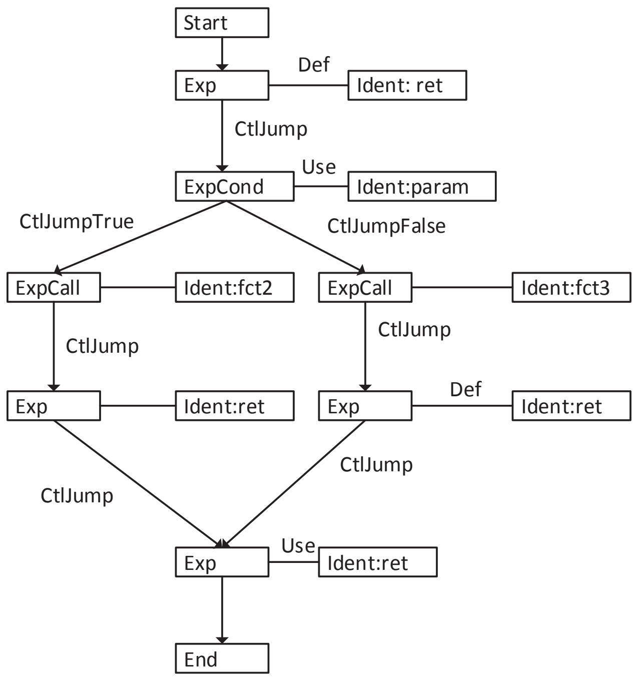
$$Kafura(s) = (KAFURA\_IN(s) \times KAFURA\_OUT(s))^2$$

where $KAFURA\_IN(s)$ is the sum of local and global incoming dataflow to the code fragment s and $KAFURA\_OUT(s)$ is the sum of local and global outgoing dataflow from s.

Mayrand *et al.* [59] presented a technique for identifying (near) duplicate functions in a large software system. The source code is first transformed into an Abstract Syntax Tree, which is subsequently transformed into an Intermediate Representation Language (IRL). For example, the function

```
int fct (int param)
{
    int ret = 0;
    if (param! = 0)
    {
        fct2();
        ret = 1;
    }
    else
    {
        fct3();
        ret = 2;
    }
    return ret;
}
```

is transformed into the following IRL representation:



The reason behind using IRL is to support multiple source languages. Moreover, the IRL abstraction carries all of the information required to compute the metrics, including control flow metrics and data flow metrics.

In total, Mayrand *et al.* conduct four points of comparison: function name, function layout, expressions, and function control flow. For function name, the symbolic names of the functions are compared. For the latter three points, Mayrand *et al.* calculate 21 metrics. For instance, for the function layout, they calculate the following five metrics: volume of declaration comments, volume of control comments, number of logical comments, number of non-blank lines, and average variable name length. By using these five layout metrics, the organization of the source code can be extracted.

Also, Mayrand *et al.* use five metrics to characterize the nature and complexity of the expressions: total calls to other functions, unique calls to other functions, average complexity of decisions, number of declaration statements, and number of executable statements.

Finally, Mayrand *et al.* use eleven metrics to capture the structure information of the control flow graph (e.g., number of decisions, number of independent paths, average decision span, etc.). When comparing two functions, Mayrand *et al.* first specify a delta threshold for each metric. Two functions are reported as similar if the absolute difference for each metric is less than or equal to the delta threshold defined for the corresponding metric.

2.2 Binary Code Reuse Detection

Text-based approaches

Text-based approaches consider the binary code as a sequence of bytes and compare the byte sequence directly.

In [40] Jang and Brumley proposed BitShred, which can identify shared code. BitShred consists of three phases: shredding a file, creating a fingerprint, and comparing fingerprints.

First, each given binary is disassembled. Once all executable code sections are identified, these sections are then divided into fragments denoted as shreds. Each shred is essentially a contiguous byte sequence of length n. The length must be appropriate to achieve a trade-off between accuracy and resistance to code reordering. An example of shredding when $n = 5$ is shown in Figure 3.

To improve scalability, a Bloom filter [23] is leveraged to create a fingerprint for each binary file. The Bloom filter is a data structure for set membership tests. It consists of a bit array of m bits and k different hash functions. To add an element to the Bloom filter, k hash functions are applied to the element, and the corresponding bits in the bit array are set to 1. For each binary file, all of the shreds are added to

```
53 8a 5c 24 08 56 24 08 56 24 08 8a 5c 24 08 56
(a) Given byte sequence

538a5c2408 8a5c240856 5c24085624 2408562408
0856240856 5624085624 2408562408 085624088a
5624088a5c 24088a5c24 088a5c2408 8a5c240856

(b) Derived shreds with size 5
```

Figure 3: An example of shredding a byte sequence when $n = 5$ (Jang and Brumley [40])

a Bloom filter which is then considered as the fingerprint of the binary file.

Jang and Brumley use the Jaccard index to measure the similarity between two files. The Jaccard index is defined as the size of the intersection divided by the size of the union of two sets [10]. However, instead of comparing two files directly, their fingerprints are compared. More specifically, Jang and Brumley use $J_R(A, B)$ to estimate the true Jaccard index between two files A and B by:

$$J_R(A, B) = \frac{S(BF_A \wedge BF_B)}{S(BF_A \vee BF_B)}$$

where $S(BF)$ returns the number of set bits of the Bloom filter BF. They also define $J_C(A, B)$ for the containment case when file A includes file B:

$$J_C(A, B) = \frac{S(BF_A \wedge BF_B)}{BF_B}$$

where $S(BF_A) > S(BF_B)$. Finally, during clustering, a similarity threshold t is defined, and two files with a similarity above t would be grouped into the same cluster.

The main problem of BitShred is that it is too coarse. Considering only the shreds of byte sequences leads to significant loss of information. Furthermore, the byte sequences are by no means stable, and could easily be changed across different binaries even when compiled from the same source code base. As a result, it is not applicable to binary code reuse detection.

FCatalog [7] is also a text-based approach for finding similarities between binary functions. FCatalog first applies k-gram analysis on the binary code to generate feature sets. Minhashing [18] is then used to convert these sets into minhash signatures of constant size. When comparing two functions, FCatalog compares their minhash signatures.

K-gram/K-perm-based approaches

Myles and Collberg [64] proposed to use opcode level k-gram as birthmarks of software. A k-gram is a contiguous substring of length k which can be letters, words, or in their approach, opcodes. Suppose $f(p)$ and $f(q)$ are two sets of k-gram birthmarks extracted from the sets of modules p and q respectively. The similarity between p and q is then defined by:

$$s(p, q) = \frac{|f(p) \bigcap f(q)|}{|f(p)|} \times 100.$$

Myles and Collberg also found that increases in the value of k result in increases in credibility, but decreases in resilience. Consequently, they claim that $k = 4$ or 5 is appropriate to achieve an acceptable trade-off between credibility and resilience.

An alternative to using k-gram is to use k-perm. In [44] Karim *et al.* used k-perm to generate phylogeny for malware. For a sequence of k characters, k-perm represents every possible permutation of that sequence. As a result, the order of characters within k-perm is irrelevant for matching purposes. Using k-perm results in the advantage of better matching of permutations of code, especially when the instructions are not in the same order. During experiments, Karim *et al.* found that using k-perm produces higher similarity scores for permuted programs. The drawback is that, for the same program, using k-perm would generates less k-perms than using k-gram. This loss of information might affect accuracy. Furthermore, apart from code reordering, there are many evolution techniques, such as instruction substitution, insertion, and deletion. Using k-perm could not track evolution in the presence of such techniques.

Metrics-based approaches

Inspired by the source code reuse detection work in [41], Sæbjørnsen *et al.* [72] proposed a practical binary code clone detection framework.

The first step of their approach is to disassemble the input binaries and extract the assembly code. The assembly code is then split into code regions using a sliding window. To allow some "fuzziness", the instructions of all code regions are normalized. Specifically, the mnemonics of the instructions are kept untouched, and the operands are normalized into three categories: *MEM*, *REG*, and *VAL*, representing memory references, registers, and constant values, respectively.

Sæbjørnsen *et al.* also defined two types of clone pairs: exact clone pairs and inexact clone pairs. Two code regions with identical normalized instruction sequences are considered as an exact clone pair. For inexact clone pairs, a similarity threshold is defined, and code regions with a similarity above this threshold are considered as inexact clone pairs.

For exact clone detection, in order to avoid pairwise comparisons of all code regions, a hashing mechanism is used to generate a fingerprint for each code region and two code regions with identical hash value are considered as a clone pair.

To detect inexact clones, Sæbjørnsen *et al.* extract some features from each code region and construct a feature vector from that. In total, the following five groups of features are extracted:

- M: Each distinct mnemonic

- $OPTYPE$: Each operand type

- $M \times OPTYPE$: Each combination of the mnemonic and the type of the first operand
- $OPTYPE \times OPTYPE$: The types of the first and second operands
- $OPTYPE \times N_k$: Each normalized operand and its index

The vectors are generated based on the number of occurrences of each feature within a code region. Subsequently, locality-sensitive hashing [18] is applied on all the feature vectors, such that similar feature vectors (code regions) are hashed into the same bucket. Then the code regions whose feature vectors are in the same bucket are considered as clone pairs. Finally, before reporting the found clone clusters, trivial clones are removed and overlapping clone pairs are merged.

Based on the framework proposed by Sæbjørnsen *et al.* [72], Farhadi *et al.* [34] designed an assembly code clone detection system named BinClone. Compared with the work in [72], BinClone can provide deterministic results and achieve better recall rates, which is of great importance in malware reverse engineering.

Similar to what Sæbjørnsen *et al.* do in [72], Farhadi *et al.* also first disassemble the binary code into assembly code, and use a sliding window to split the assembly code into code regions. During normalization, unlike Sæbjørnsen *et al.* who only normalize the operands into three categories *MEM*, *REG*, and *VAL*, Farhadi *et al.* use a more fine-grained approach with different levels of hierarchy. In Farhadi *et al.*'s approach, operands are first normalized to three categories *MEM*, *REG*, and *VAL*. The *REG* category is further normalized into 3 groups: General Registers (e.g., EAX, EBX), Segment Registers (e.g., CS, DS), and Index and Pointer Registers (e.g., ESI, EDI). Finally, the General Registers group is broken down by size into three groups: 32-bit registers (e.g., EAX), 16-bit registers (e.g., AX, BX), and 8-bit registers (e.g., AH, BL).

For exact clone detection, Farhadi *et al.* use a similar hashing mechanism as used by Sæbjørnsen *et al.*

For inexact clone detection, Farhadi *et al.* proposed two methods, a sequential feature selection method and a two-combination method. The sequential feature selection method first computes the median of each feature on all regions and filters out features whose median equals to 0. A binary vector is then generated for each region by comparing the feature vector of the region with the median vector. Then the binary vector is partitioned into sub-vectors which are hashed into different buckets. By keeping track of the frequency of region co-occurrences of all buckets, inexact clones are identified.

The two-combination method is similar to the sequential feature selection method. However, the two-combination method considers all possible two-combinations of features when generating sub-vectors. Consequently, the set of sub-vectors generated by the sequential feature selection method is a subset of the sub-vectors generated by the two-combination method. As a result, the two-combination method has a better recall rate. However, there is a trade-off of lower performance of scalability than the sequential feature selection method.

The problem is, their approach does not take the structure of assembly function into account. Instead, it partitions each function into multiple code regions and matches these code regions. As a result, the precision is a problem.

In [25] Bruschi *et al.* proposed a novel metric-based approach for detecting self-mutating malware. Virus writers may reuse code when creating malware. In order to evade anti-virus products, they may introduce various mutation techniques, such as instruction substitution, instruction permutation, dead code insertion, variable substitution, and control flow alteration.

To overcome these evasion techniques, Bruschi *et al.* propose to first normalize the code to a canonical form, which is suitable for comparison, and to then compare the normalized code. The following normalization techniques are used:

- Instruction meta-representation: This is a high level representation of the semantics of the original machine instructions. All side effects on registers, memory, and control flags are recorded in this representation. For example, "`pop eax`" will be translated into "`r10 = [r11]; r11 = r11 + 4`".

- Propagation: This is used to propagate forward values assigned or computed by intermediate instructions. Once an instruction defines a value, its occurrences would be replaced with the value computed by the defining instruction, if subsequent instructions did not redefine it. By doing this, all temporary variables can be eliminated and higher level expressions can be generated.

- Dead code elimination: This technique removes instructions whose results are never used.

- Algebraic simplification: This simplifies expressions according to ordinary algebraic rules.

- Control flow graph compression: This technique analyzes inserted fake conditional and unconditional jumps. If the condition of a jump always evaluates to true or false, which means that the underlying path will never be accessed, then all the paths originating from it should be removed.

After all the codes have been normalized, Bruschi *et al.* begin to compare them and measure the similarity between them. To do this, the approach proposed by Kontogiannis *et al.* [50] is adopted. More specifically, every control flow graph is encoded into a 7-vector using the metrics shown in Table 1:

For each given code fragment, a 7-vector $(m_1, m_2, m_3, m_4, m_5, m_6, m_7)$ is generated as its fingerprint. To compare two code fragments a and b, the Euclidean distance

Metrics for encoding a control flow graph into a vector
m_1: number of nodes in the control flow graph
m_2: number of edges in the control flow graph
m_3: number of direct calls
m_4: number of indirect calls
m_5: number of direct jumps
m_6: number of indirect jumps
m_7: number of conditional jumps

Table 1: Metrics (Bruschi *et al.* [25])

between their fingerprints is calculated as:

$$\sqrt{\sum_{i=1}^{7}(m_{i,a} - m_{i,b})^2}$$

where $m_{i,a}$ and $m_{i,b}$ are the i^{th} metric calculated on fragments a and b, respectively. Bruschi *et al.* also define a threshold, and if the calculated distance is below this threshold, these code fragments are considered equivalent.

Structure-based approaches

Flake [35] and its extension proposed by Dullien and Bochum [32] presented a pioneer work of structure-based comparison approach. Their work aimed at comparing two different but similar executables. The whole approach is based on the observation that the call graph of an executable stays largely the same, even when compiled in different compilation environments. They also introduced a novel way of comparing two basic blocks, namely small prime product (SPP), which is resilient to instruction reordering.

The SPP algorithm works as follows. First, a unique small prime number is assigned to each distinct mnemonic based on an arbitrary but deterministic order. Since every basic block can be looked at as a sequence of instructions, then the product of all corresponding primes based on the mnemonics of consisting instructions is calculated. If two basic blocks have an identical product, they must have the same set of mnemonics, though not necessarily in the same order due to the uniqueness of prime decompositions and the fact that multiplication is commutative. For real-world applications, this product is truncated (module 2^{64}). Dullien and Bochum also proved that this truncation is safe for a sequence shorter than 14 elements with an alphabet of 100 elements. Naturally, this SPP can also be applied to compare two functions as well.

Given two executables, the call graphs of both executables are first constructed, followed by the generation of control flow graphs for all functions of both executables. Note that each node in the call graph is essentially a function that can be replaced by its control flow graph. Dullien and Bochum operate first on call graphs. More specifically, they generate a number of "fixedpoints" in the call graphs of both executables by selecting node (function) pairs that meet the following criteria:

- K-indegree nodes / k-outdegree nodes: Selecting functions whose indegree or outdegree is exactly k
- Recursive nodes: Selecting functions that invoke themselves
- Same name: Selecting functions with the same symbolic name
- Same string reference: Selecting functions that contain code referring to the same string
- Same SPP: Selecting functions that have the same prime product

A 3-tuple is also assigned to each node (function) representing three extracted features: the number of basic blocks in the function, the number of edges linking them to form the control flow graph, and the number of subfunction calls in all consisting basic blocks of that function. This 3-tuple is then considered as a feature vector. When comparing two nodes, the Euclidean distance between their feature vectors is calculated.

After the initial fixedpoints have been generated, Dullien and Bochum continue to explore their successors and predecessors to expand these fixedpoints. Given an unmatched node in one executable, the node with the minimal distance from the other executable is chosen. If multiple nodes with the same minimal distance to that unmatched node are found, then no nodes will be chosen.

More formally:

$$s_c(x, A) := \begin{cases} a \texttt{ if } \exists_{a \in A} \forall_{b \in A, b \neq a} |x - a| < |x - b| \\ 0, \; \mathit{Otherwise} \end{cases}$$

After all fixedpoints are generated, the output is a (partial) mapping of functions in both executables. For each matched function pair, Dullien and Bochum begin to work on their control flow graphs in the same manner as in call graphs. The fixedpoints in control flow graphs are generated using the following criteria:

- K-indegree nodes / k-outdegree nodes: Selecting basic blocks that have the same number of predecessors or successors
- Recursive nodes: Selecting basic blocks that may jump back to their beginning
- Same string reference: Selecting basic blocks that contain code referring to the same string
- Same SPP: Selecting basic blocks that have the same prime product
- Same subfunction call: Selecting basic blocks that contain calls to the same subfunction

The output of the control flow analysis is a (partial) mapping of basic blocks of every matched function pair. Once all the matchings have been generated, every instruction in a basic block is then treated as a node and every basic block is treated as a special simple form of graph, and again, the same algorithm used on call graphs and control flow graphs is applied on basic blocks.

Dullien and Bochum applied their approach on two well-known trojans: `Bagle.X` and `Bagle.W` [2]. A thorough analysis of `Bagle.X` had already been conducted, with every function properly named and commented. By using their approach on both the new found `Bagle.W` and already analyzed `Bagle.X`, Dullien and Bochum successfully associate all but 6 functions of `Bagle.W` with their counterpart in `Bagle.X`. As a result, human analysts have only to focus on these 6 unmatched functions, which greatly saves the analysts both effort and time.

Despite encouraging results, the largest shortcoming of this approach is that it is doing exact matching, instead of fuzzy matching. For example, when generating fixedpoints for control flow graphs or for call graphs, this approach requires two nodes to have the same in-degree or out-degree. In addition, the same SPP requires two basic blocks (or functions) to have exactly the same set of mnemonics, which do not always hold true even for a true match. These requirements are overly strict, especially for executables compiled in different compilation environments. As a result, this approach is only suitable for comparing the same executable, from the same source code base, and compiled in the same compilation environment, which greatly limits its usefulness.

BinDiff [3], developed by the Zynamics company, is the de facto standard commercial tool for comparing binary files. The intention of BinDiff is to compare two related and similar, but different executables, and to identify identical or similar functions among them. One advantage of BinDiff is that it is applicable across various platforms. Analysts can apply BinDiff on binary files from different platforms, such as x86, MIPS, ARM, and PowerPC. The changed (unmatched) functions and basic blocks are displayed in an easy to understand way.

In [31], Dullien *et al.* presented some results on executable code comparison for attacker correlation. After a successful attack has taken place, the only thing left for defenders to analyze is the malicious code obtained from compromised systems. Dullien *et al.* thus focus their analysis on the structural features, such as the call graph and control flow graph, of the malicious code. Since using pairwise comparisons to compare one piece of malicious code with a large repository of code does not scale well, Dullien *et al.* designed a way to encode the control flow graph into a sequence of bits to allow fast querying into large sets of data. To do this, every edge in a control flow graph is first converted into an n-tuple of integers using the following function:

$$tup : \mathfrak{G} \to \mathfrak{P}(\mathbb{Z}^5)$$

$$tup(g) \mapsto \left\{ \left(\begin{array}{c} topologicalorder(src(e)), \\ indegree(src(e)), \\ outdegree(src(e)), \\ indegree(dest(e)), \\ outdegree(dest(e)) \end{array} \right) | e \in E_g \right\}$$

where $\mathfrak{G}$ is the set of all control flow graphs and E_g is the set of edges of a particular G belonging to $\mathfrak{G}$. Next, each 5-tuple is converted into a real number as follows:

$$emb(z) \mapsto z_0 + z_1\sqrt{2} + z_2\sqrt{3} + z_3\sqrt{5} + z_4\sqrt{7}.$$

Finally, the *MD-Index* for a given graph is calculated using a hash function:

$$Hash(g) = \sum \frac{1}{\sqrt{emb(t)}}$$

where $t \in tup(g)$.

Dullien *et al.* also designed an algorithm that operates at both call graph and control flow graph levels, to construct an approximation of the maximum subgraph isomorphism. The algorithm attempts to match nodes and edges based on the following characteristics:

- Bytes Hash: A traditional hash over the bytes of the function or basic block
- MD-Index of a particular function, which is the MD-Index of the underlying function of the given node
- MD-Index of the source and destination of edges of the call graph, which is a tuple consisting of the MD-Index of the source node and destination node
- MD-Index of the graph neighborhood of a node/edge, which is the MD-Index of a subgraph containing the given node/edge, extracted from the original graph
- Small Prime Product, which is a simple way of comparing the mnemonics of a basic block or function [32]

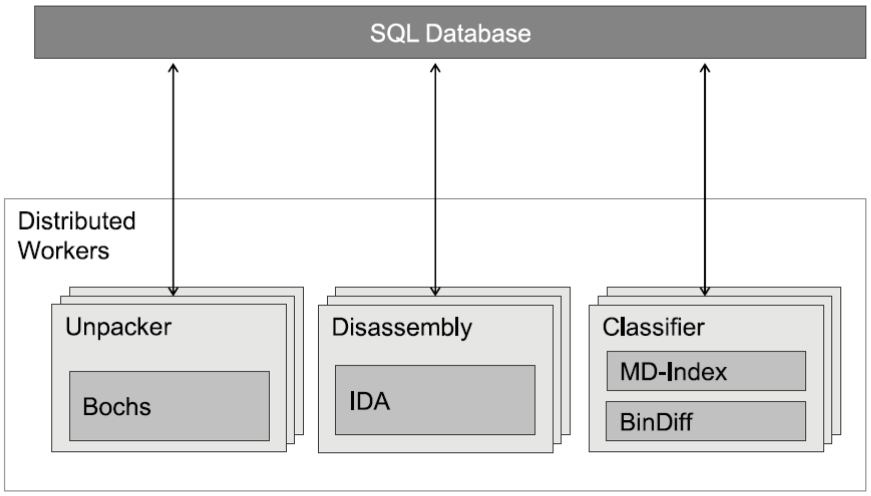


Figure 4: System architecture (Dullien *et al.* [31])

Figure 4 depicts the overall system architecture. The central part is an SQL database from which multiple components fetch data. In total, there are four components:

- *Unpacker*. This component attempts to remove the encryption by emulating the executable and monitoring the statistical properties of the system RAM. Once the entropy of memory drops, the component assumes that the encryption was removed.

- *Disassembly*. This component disassembles the memory dump from the *Unpacker* and extracts call graphs and control flow graphs.

- *Scheduler*. The *Scheduler* conducts a rough comparison based on the MD-Indices of the functions in the disassembly and obtains a subset of promising executables for later comparison.

- *Comparison*. This component performs the comparison and writes the results back to the database.

Kruegel *et al.* [53] presented a novel technique based on structural analysis to detect polymorphic worms. Polymorphic worms are able to change their code while spreading. To detect such kind of worms, first control flow graphs are constructed from the network stream. Then, k-subgraphs are generated from the control flow graphs. To do this more efficiently, a depth-first traversal is conducted on each basic block b to first generate the spanning tree. Once the spanning tree is constructed, all possible k-node subtrees are generated with basic block b as the root node. To ease the comparison of two subgraphs, canonical graph labeling [19] is leveraged to transform every subgraph into its canonical representation so that two subgraphs with identical canonical forms are isomorphisms. To overcome the limitation of considering only the structure, every basic block is assigned a color based on the instructions, to encode the functionality of that basic block. The color value used by Kruegel *et al.* is 14-bit. Correspondingly the assembly instructions are classified into 14 categories and each bit in the color value represents the presence of a certain category of instructions. The Nauty library [61] is then used to take the color into consideration when canonicalizing a subgraph. By combining the structural analysis and graph coloring techniques, the high-level structure of a polymorphic worm is captured.

In [24], Bruschi *et al.* proposed an approach for malicious code detection. Given a malicious code M and a program P, Bruschi *et al.* first collect information from P by conducting control flow and data flow analysis. This information is then used to normalize the program to P_N, after which the control flow graphs for the malicious code and P_N are generated. To augment these control flow graphs, labeling (coloring) is applied to both nodes and edges based on instruction classes and flow transition classes, respectively. Bruschi *et al.* label nodes in a manner very similar to the work proposed in [53], but with fewer instruction classes. Subsequently, a subgraph isomorphism algorithm, VF2 [27], is applied on the labeled control flow graphs. The corresponding isomorphism in P_N is then reported as an instance of the malicious code, as depicted in Figure 5.

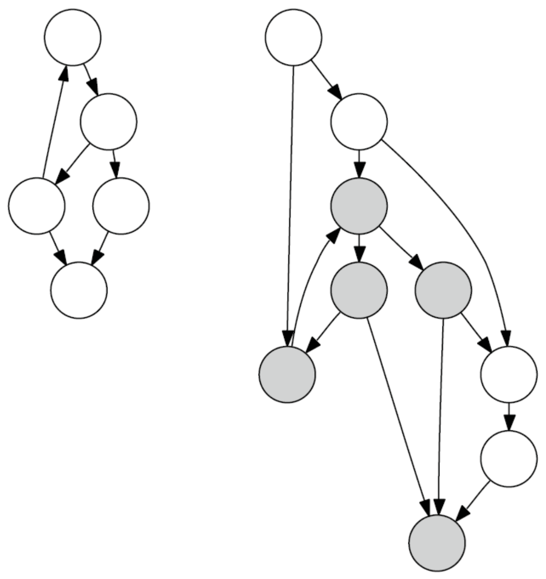


Figure 5: Malicious code M and normalized program P_N, respectively. The highlighted nodes are the matching nodes in P_N (Bruschi *et al.* [24])

Semantic-based approaches

The semantic-based approach bases its analysis on the semantics of the binary code. In the case of binary code reuse detection, semantics often refer to the input-behavior and output-behavior of the binary code. Semantic-based approaches are often combined with structural analysis or with clustering methods from machine learning.

BinHunt [36], proposed by Gao *et al.*, is the first work that combines symbolic execution and theorem proving to perform binary code similarity comparison. Figure 6 depicts the overall architecture of BinHunt. Given two binary files, the first step is to disassemble both files. In their implementation, a commercial disassembler, IDA Pro [9], is used. Note that Gao *et al.* use IDA Pro only to obtain a sequence of x86 instructions; they do not rely on it to generate the control flow graph. Subsequently, all x86 instructions are fed to a converter, and the output is an intermediate representation (IR) of the original x86 instructions. This IR is far simpler than the original instruction set. It consists of roughly a dozen different statements that are type-checked and free of side effects. The generated IR is then fed into a control flow graph constructor to generate the control flow graph for each function as well as the call graph for each binary. The control flow graph of an assembly function is a directed graph, where nodes represent basic blocks and edges represent the execution flow between basic blocks. To compare two control flow graphs, Gao *et al.* first introduce a way to compare their nodes (basic blocks) in terms of semantics.

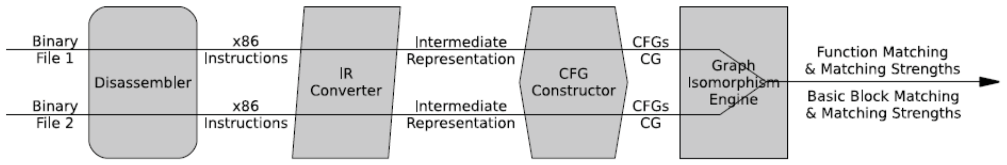


Figure 6: Overall architecture of BinHunt (Gao *et al.* [36])

For every basic block, Gao *et al.* first find all input and output registers and variables from the IR representation. Then, symbolic execution [48] is used to obtain the final values of all output registers and variables. After that, a theorem prover is applied to test if two basic blocks have the same output registers and variables. Gao *et al.* claim that two basic blocks are functionally equivalent if there is a permutation of all the registers and variables between the two basic blocks such that all the matched registers and variables contain the same value. A matching strength is also assigned to every matching basic block pair to denote the similarity of the two basic blocks in terms of functionality.

Guided by this matching strength, subgraph isomorphism analysis is conducted. More specifically, when comparing two functions, the backtracking [75, 52] algorithm is used to find the maximum common subgraph of the control flow graphs of these two functions. The backtracking algorithm keeps the best match found so far and replaces erroneous matches with a better one until the best one has been found. The output is four fold, namely the (partial) matching between functions, the (partial) matching between basic blocks for matched functions and the matching strengths for matched functions and basic blocks.

To demonstrate the effectiveness of BinHunt, Gao *et al.* applied BinHunt on three use cases. The first use case corresponds to a buffer overflow vulnerability in `gzip`. By comparing both the patch and unpatched version of `gzip`, BinHunt successfully identified the function where the patch took place. Gao *et al.* also conducted experiments on other executables such as `tar` and Microsoft.NET framework 2.0 (`ASP.NET`), and successfully identified these functions whose functionality has been changed.

Despite encouraging results, BinHunt is not without drawbacks. The first problem is that using graph isomorphism to detect similar binary code is overly strict and is not suitable for practical use. Its usefulness is severely limited by the fact that compilers may bring mutations or noise to the control flow structure. As a result, BinHunt is only suitable for analyzing two versions of the same binary compiled by the same compiler in the same compilation environment. Second, BinHunt focuses its analysis on x86 executables, as the used IR converter does not support x64 or other platforms. Third, symbolic execution, theorem proving, and graph isomorphism detection are all time-consuming. For example, during graph isomorphism, a timeout must be introduced for function pairs in which the backtracking algorithm did not return in a timely fashion and no results will be returned in such case. It is thus not suitable for large-scale code reuse detection.

Recently proposed by Ng and Prakash [65], Exposé is a tool for identifying library code reuse in applications. It combines symbolic execution using theorem proving with k-gram at the function level to achieve a trade-off between performance and accuracy. To do this, Exposé divides the functions of an application into two sets based on numerous criteria. Symbolic execution [48] is then performed on one set, while k-gram analysis is performed on the other.

The first step is pre-filtering, during which loader support functions are excluded. Exposé also excludes functions that are improbable for semantic equivalence checking by only selecting functions with the following criteria:

- Has the same number of input arguments as the target library function.
- Has the same out-degree as the target library function.
- With a cyclomatic complexity of less than 15.
- With a function size of less than 300 bytes.

These criteria are required due to the fact that checking semantic equivalence using symbolic execution is very expensive. Excluding functions that are unlikely to be matched or are too expensive to match could speed up the process.

Exposé also maintains two sets, an `IS`-pairs set and a `MAY`-pairs set. If two functions from the library and executable respectively are semantically equivalent, Exposé places them into the `IS`-pairs set. For functions that are improbable for semantic equivalence checking, k-gram analysis is applied. First, a feature vector is generated for each function. Then, to compare two functions, the cosine distance [4] between the feature vectors of these two functions is calculated. After that, Exposé biases the original cosine distance following a specially crafted score strategy. For example, if two functions have different out-degrees, Exposé increases the cosine distance by 0.1, and if they have the same number of non-zero input parameters, Exposé decreases the distance by 0.2. Ng and Prakash claim that the values used for biasing work well in their experiments. Then, for every library function, Exposé selects five functions with the smallest biased cosine distance from the application, and places them in the `MAY`-pairs set.

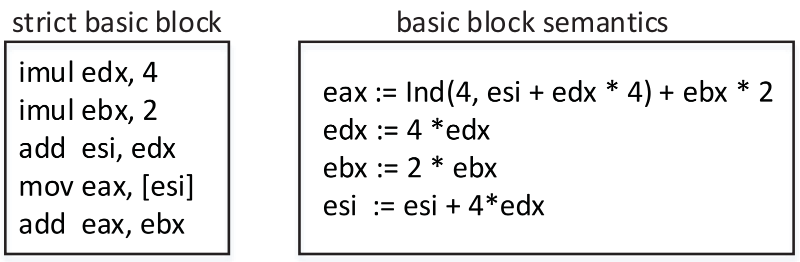
Once all `IS`-pairs and `MAY`-pairs have been obtained, Exposé uses the Hungarian algorithm [54] to find the best localized mapping of functions between the library and the application. The final distance score between the library and the application is the average of the biased cosine distance of the pairs returned by the localized Hungarian algorithm. The smaller the score, the more likely the application contains the code of that library.

Ng and Prakash conducted two experiments to evaluate Exposé. In the first experiment, given the library `libpng` and an application known to contain the code of `libpng`, Exposé correctly ranked the application as #1 out of 128 applications. In the second experiment, using `zlib` as the target library, Exposé successfully identified 10 applications that are known to contain the target library from an assortment of 2,927 applications.

Since Exposé aims to rank applications based on the likelihood of containing the target library code, the features it uses, such as k-gram, are very coarse and are not suitable to accurately compare two binary functions. Moreover, if an application reuses part of the library's code, Exposé may fail to detect this kind of partial code reuse.

More recently, Pewny *et al.* proposed TEDEM [67], which is a binary code reuse detection system which can identify the buggy function from a set of reference functions. Unlike previous works which leverage theorem proving, Pewny *et al.* designed a novel way of comparing code regions semantically by leveraging tree edit distance.

Similar to most existing works, the first step is to disassemble the buggy function and all reference functions and to generate the control flow graph with nodes representing basic blocks. However, each basic block is further split into strict basic blocks at function calls. Pewny *et al.* argue that the instruction following a call is actually an implicit target of a return instruction, and thus should be the first instruction of a strict basic block.



```
imul edx, 4
imul ebx, 2
add  esi, edx
mov eax, [esi]
add  eax, ebx
```

```
eax := Ind(4, esi + edx * 4) + ebx * 2
edx := 4 *edx
ebx := 2 * ebx
esi  := esi + 4*edx
```

Figure 7: Exemplary strict basic block and its semantic equations, where $Ind(x, y)$ refers to the x-byte value at address y

After all (strict) basic blocks have been obtained, Pewny *et al.* use METASM [37] to extract the semantics of each basic block. METASM, written in Ruby, is capable of executing or accumulating assembly instructions. The output of the execution is a list of equations which represent the effect of the execution on registers, memory, or branch conditions. Figure 7 shows an example of a strict basic block and its semantic equations extracted by METASM. Subsequently, all semantic equations are converted into S-Expressions. For example, the S-Expression of the first equation in Figure 7 would be:

$$(:= \ eax \ (+ \ (Ind \ 4 \ (+ \ esi \ (* \ edx \ 4))) \ (* \ ebx \ 2 \ ))).$$

Since the S-Expression is a notation for tree-like data structures, every equation is then represented by a tree, such that the root node is an assignment, the leaf nodes are registers or constants, and the intermediate nodes are operations. To compare two basic blocks, Pewny *et al.* first transform all semantic equations of both basic blocks into trees, and then use the tree edit distance (TED) to measure the distance of these two basic blocks. To this end, the tree edit distance algorithm proposed by Tekli *et al.* [74] is adopted. Pewny *et al.* choose this algorithm specifically for its ability to support subtree-edits. An example is shown in Figure 8.

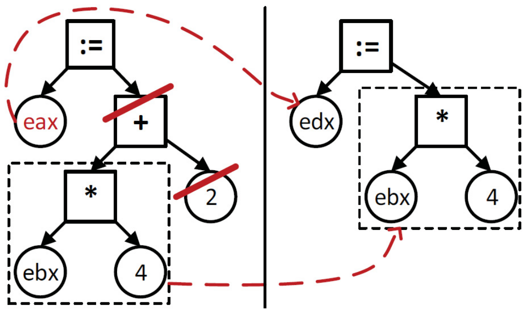


Figure 8: Exemplary tree edit distance with subtree-edits (Pewny *et al.* [67])

Basic block comparison using TED is the cornerstone of their approach. However, comparing basic blocks using TED does not scale well. To address this problem, given a target buggy function, Pewny *et al.* first use coarse-grained basic block features to find a small set of candidate basic blocks from the reference functions. The features they use include the number of equations, the depth of the equation trees, and the number of nodes in the tree. This small set is then used as a set of starting points. Then, for every starting point, Pewny *et al.* use the fine-grained feature, namely TED, to further select 20 basic blocks with the smallest tree edit distance from the reference functions as matching candidates.

For every starting point and each of its 20 matching candidates, Pewny *et al.* conduct neighborhood exploration in both the buggy function and the reference function based on their control flow graphs. Each time, newly matched basic block pairs will be used to expand the existing mapping. The exploration terminates when no further neighbors can be explored. Pewny *et al.* conduct this for each matching candidate of each starting point. The final distance is the smallest sum of tree edit distances of all the matching basic block pairs found so far. The smaller the distance, the more similar the reference function is to the buggy function.

The largest limitation of this approach is performance. Recall that for every basic block in the set of starting points, 20 matching basic blocks must be found using tree edit distance. It is thus not suitable for code reuse detection in large data sets. Moreover, this approach must conduct neighborhood exploration multiple times and each time it starts exploration from only one pair of basic blocks. As a result, the quality of the mapping found by this approach might not be good.

Behavioral-based approaches

Comparetti *et al.* [26] developed a system named *REANIMATOR* which can determine the capabilities of malware programs by dynamically executing the malware and simultaneously observing its behavior. When malicious actions are observed during dynamic execution, Comparetti *et al.* extract and model the parts of the malware binary that caused this behavior. These models are then used to check whether similar code is present in other malware samples. Their system consists of three phases: dynamic behavior identification, extracting genotype models, and finding dormant functionality. In the dynamic behavior identification phase, a dynamic execution sandbox, Anubis [1], is leveraged to execute the given malware binary and to record all invocations of security-relevant system calls and Windows API functions. In the second phase, the part of the binary responsible for certain behavior is located and modeled. Finally, all the models built in the previous phase are used to check other binaries for dormant functionality.

In [73], Shankarapani *et al.* proposed two methods, Static Analyzer for Vicious Executables (SAVE) and Malware Examiner using Disassembled Code, for malicious code detection. The first method, SAVE, focuses on behaviors of API calls (e.g., static API call sequence and static API call set) for analysis, whereas the second method focuses analysis on assembly calls of the code.

Hybrid detection approaches

In [47], Khoo *et al.* built a search engine for binary code in which they combined five different abstraction techniques: instruction mnemonic k-grams, instruction mnemonic k-perms, control flow subgraph, extended control flow subgraph, and data constants. Khoo *et al.* first generate mnemonic k-grams [64] from a given piece of binary code. Since the mnemonic k-grams are not resilient to instruction reordering, mnemonic k-perms [44] are combined together with k-grams. To capture control flow structure information, Khoo *et al.* first break the control flow graph into several small k-subgraphs, and then use graph canonicalization [61] to transform each k-subgraph into a k^2-bit number in order to ease the comparison of graphs. To address the shortcoming of low uniqueness of k-subgraphs, Khoo *et al.* propose to use extended control flow subgraph by introducing a virtual external node. Finally, two types of data constants, integers and strings, are extracted from the binary code.

Lastly, Wang *et al.* [76] presented a tool called *BMAT*, which can match two versions of a binary program without knowledge of source code changes. The intention is to propagate the profile information from an old build to a newer build, and thus, save the time of re-profiling.

2.3 Summary

We have presented different approaches for both source code reuse detection and binary code reuse detection. Unlike the source code, the binary code is essentially a sequence of assembly instructions with mnemonics, various registers and memory references. All these mnemonics, different registers and memory references are very abstract, and do not preserve much information from the source. Consequently, these source code reuse detection techniques cannot be applied to binary code reuse detection. On the other hand, existing binary code reuse detection techniques are not without their limitations. For example, these text-based approaches are too coarse, and are not suitable for function level binary code reuse detection; the metrics-based approaches normally can handle a large collection of code, but the precision is a problem as different code might have identical or similar feature vectors. Structure-based approaches take the control flow graph or the call graph into consideration; however, it is relatively expensive to compare two graphs especially for large graphs. Compared with other techniques, semantic-based approaches can achieve better accuracy. However, using semantics to compare binary code is expensive and overly strict. We need an approach that is efficient, accurate and at the same time scalable.

Chapter 3

Algorithm Description

The problem we are trying to solve can be described as follows: Given one target binary function from one executable, and a large repository with thousands or millions of functions from other executables, how to identify all the identical or similar functions from the repository. This problem is two-fold. First, how to compare two assembly functions and obtain a similarity score. Second, how to efficiently retrieve those ones that are likely to be identical or similar to our target function and at the same time, avoid pairwise comparison of each function pair.

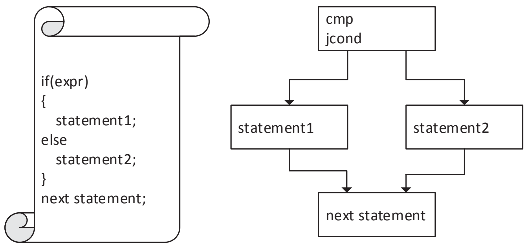
In this work, we establish the similarity of two functions by comparing their control flow graphs (CFGs). The CFG of an assembly function is a directed graph, where nodes represent basic blocks, and edges represent the execution flow between basic blocks.

The compiler is responsible for transforming the source code into assembly code. Take C++ for example, generally speaking there are four types of control structures:

- Sequential control structure
- Selection control structure (e.g., if, if-else or switch statement)
- Iteration control structure (e.g., for, while or do-while loop)

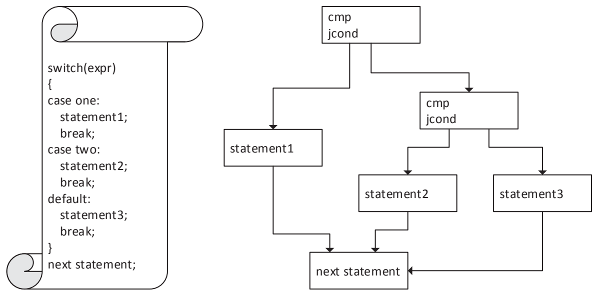
- Goto structure

Normally the sequential control structure will not bring additional edges or branches into the control flow graph, while the later three structures would. Figure 9 shows some typical examples of these structures and their corresponding CFGs. Note that as these structures can be nested in source code, so do their corresponding CFGs. Figure 10 depicts two examples of nested control structures.



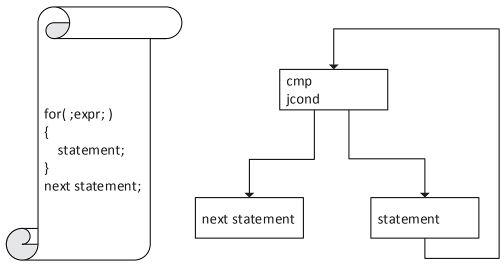
```
if(expr)
{
   statement1;
else
   statement2;
}
next statement;
```

(a) The "if-else" structure



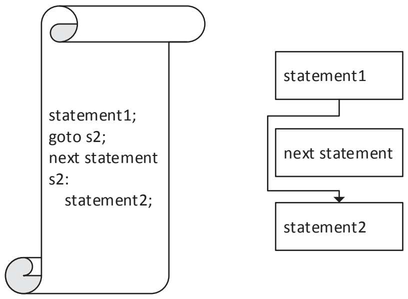
```
switch(expr)
{
case one:
   statement1;
   break;
case two:
   statement2;
   break;
default:
   statement3;
   break;
}
next statement;
```

(b) The "switch" structure



```
for( ;expr; )
{
   statement;
}
next statement;
```

(c) The "for" loop structure



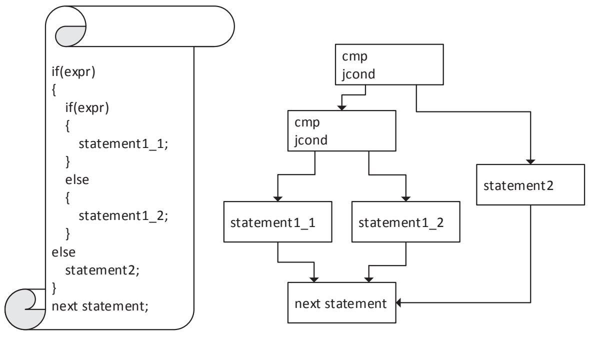
```
statement1;
goto s2;
next statement
s2:
   statement2;
```

(d) The "goto" statement

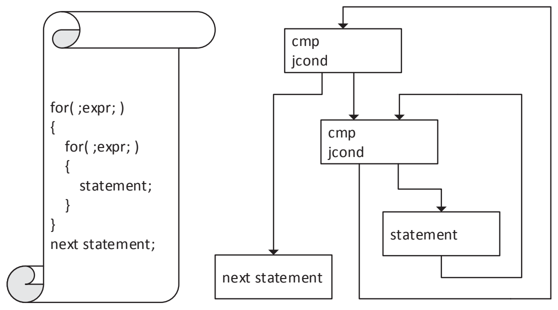
Figure 9: Examples of control structures and corresponding CFGs

Although different compilation environments would bring some mutations or "noise" into the CFGs, still the overall structure is relatively stable. As can be seen from Figure 9, the mapping between source code statements and basic blocks is stable as well.

Based on these observations, we choose to use a basic block-centric approach when



(a) The nested "if-else" structure



(b) The nested "for" loop structure

Figure 10: Examples of nested control structures and corresponding CFGs

comparing two functions. We first find the mapping of basic blocks between these two functions and then for every matching basic block pair, we obtain a matching score using the concept of longest common subsequence. Finally, we calculate the similarity score of two functions from the matching results of the basic blocks.

Since pairwise comparison is not efficient, we choose to apply a filtering process before the actual comparison. Given a target function, instead of comparing it with every function in the repository, we choose to first obtain a subset of promising functions using the filtering process and then pairwise compare the target with every function in this subset.

The rest of this chapter is organized as follows. First, the overall design and workflow of our approach is presented in Section 3.1. Then we introduce every step of the algorithm we use to compare two functions, namely disassembly and normalization (Section 3.2), instruction comparison (Section 3.3), basic block comparison (Section 3.4), longest path generation (Section 3.5), path exploration (Section 3.6) and neighborhood exploration (Section 3.7). In Section 3.8 we introduce the detailed design of our filtering process and Section 3.9 summarizes this chapter.

3.1 Overview

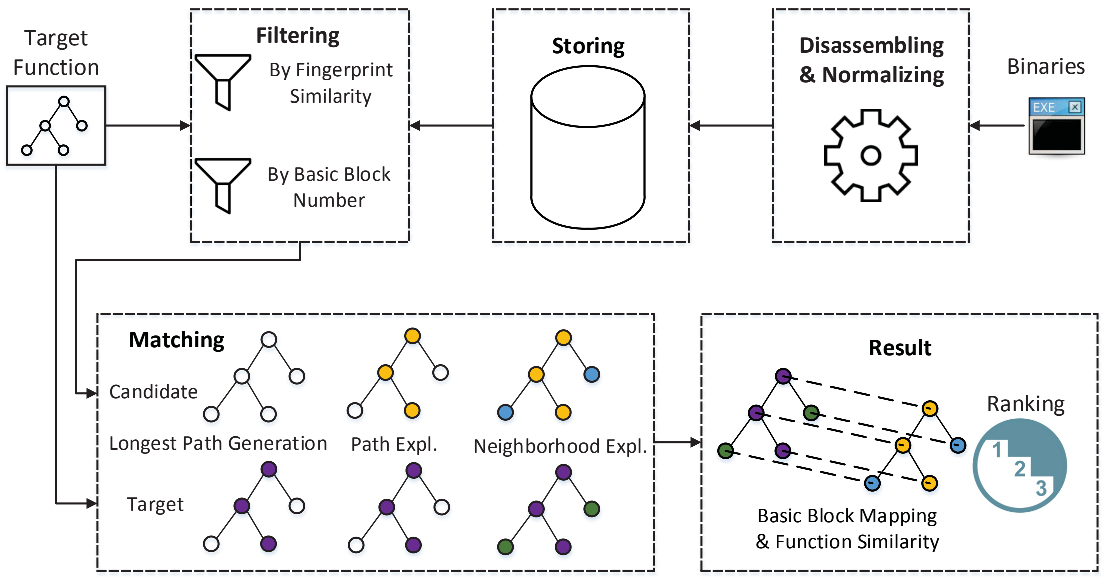


Figure 11: Workflow of BinSequence

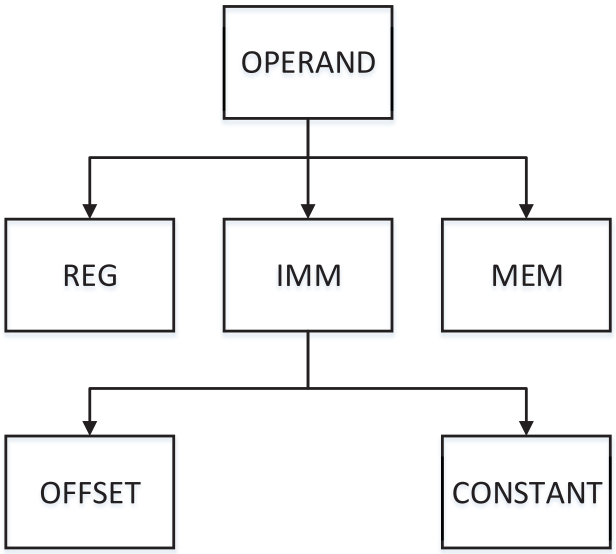
Figure 11 depicts the workflow of BinSequence. First, a collection of interesting binaries such as previously analyzed malware or open source software that may have been reused, is disassembled. The output is a set of functions. We then keep all the functions in a large repository after normalizing them. Given a target function, we can compare it with every function in the repository and rank the results. However, this is not efficient as most of the functions in the repository are not similar to our target and should thus not be compared. To speed up the process, we focus only on those functions that are likely to be similar with our target. To this end, we adopt a filtering process in which we use two filters. The first filter is based on the number of basic blocks, while the second is based on the similarity of feature sets that we extracted as fingerprints for every function. The output of the filtering process is a subset of functions from the repository, which we call the candidate set. We then perform pairwise comparisons of the target function with every function in the candidate set. The comparison consists of three phases. First, we generate the longest path of the target function. Then we explore the reference function in the candidate set to find the corresponding matching path, from which we can obtain the initial mapping of basic blocks. We then improve the mapping through neighborhood exploration in both the target and reference functions. The output is the mapping of basic blocks and the similarity score of these two functions. After we have done this to every function in the candidate set, we obtain a ranking of functions based on the similarity score.

3.2 Disassembly and Normalization

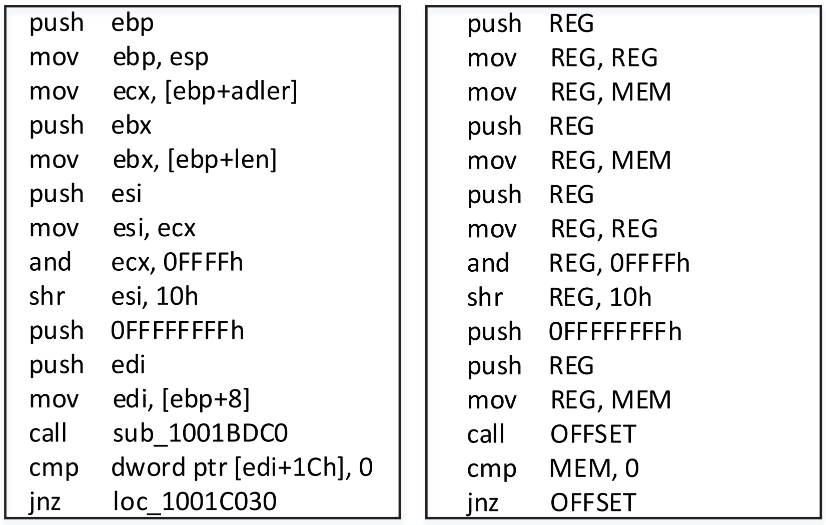
Given a collection of binaries, the first step is to disassemble each binary to a set of functions. In our experiments, we use IDA Pro [9] to generate the control flow graph for every function. Since the compiler has many choices with regard to mnemonics, registers and memory allocations when generating the assembly code, it is essential that every assembly instruction in the basic block is normalized before comparison [72].

Note that an assembly instruction consists of a mnemonic and a sequence of up to 3 operands. When normalizing instructions, we keep the mnemonics untouched, and only normalize the operands. We classify the operands into three categories, namely registers, memory references and immediate values. For immediate values, we further normalize them into two categories, memory offsets (addresses) and constant values as depicted in Figure 12a. The reason to differentiate between addresses and constant values is that addresses would change according to different assembly code layouts while constant values do not. If an immediate value is classified as constant value, we keep the literal value. The motivation is that normally constants stay the same even when different compilers or optimization levels have been used.

Some literatures also consider strings as a special type of data constants [47, 29].



(a) Normalization hierarchy



```
push   ebp
mov    ebp, esp
mov    ecx, [ebp+adler]
push   ebx
mov    ebx, [ebp+len]
push   esi
mov    esi, ecx
and    ecx, 0FFFFh
shr    esi, 10h
push   0FFFFFFFFh
push   edi
mov    edi, [ebp+8]
call   sub_1001BDC0
cmp    dword ptr [edi+1Ch], 0
jnz    loc_1001C030
```

```
push   REG
mov    REG, REG
mov    REG, MEM
push   REG
mov    REG, MEM
push   REG
mov    REG, REG
and    REG, 0FFFFh
shr    REG, 10h
push   0FFFFFFFFh
push   REG
mov    REG, MEM
call   OFFSET
cmp    MEM, 0
jnz    OFFSET
```

(b) An exemplary basic block and its normalized version

Figure 12: Basic block normalization

In [29] David and Yahav replace an offset with the string and take the string into comparison if the offset points to a string. But we consider only integers. The reason is that strings can easily be modified without difficulty. A malware author could easily evade those string based detection techniques by changing the strings inside the source code without changing the functionality. However, the integers are more related to the functionality, which makes them a better target in reverse engineering.

3.3 Instruction Comparison

Inspired by the recent work in [29], we use a similar strategy when comparing instructions. As depicted in Algorithm 1, for two normalized instructions, if they have different mnemonics, then their matching score is 0 regardless of their operands. Otherwise, we give them a score for identical mnemonic and continue to compare their operands. If their corresponding operands are the same after normalization, then we give them an additional score for each matching operand. Notice that mnemonics represent the low-level machine operations and carry more information than operands, thus we should give a higher score to identical mnemonic. At the same time, to avoid the information carried by operands from getting neglected, this score could not be overly high. Constants also carry much information from the source. When comparing two constant operands, we further compare their literal values. If their literal values are the same, we then give them an additional score. During our experiments we found that it is appropriate to give score 1, 2 and 3 to identical operand, mnemonic and constant respectively. Using these score values, we can allow those important parts of instructions to match, and at the same time, without getting misled by this score strategy. Following this strategy, we can calculate that the score of comparing `push eax` with `push ebx` is 3, as both are `push REG` after normalization, while the score of comparing `push 0` with `push 1` is only 2 as the literal value of their operands is not the same.

Algorithm 1: Compare two instructions

```
Input: Two normalized instructions
Output: The matching score of two instructions
Function CompIns(ins1,ins2)
    score = 0
    if ins1.Mnemonic == ins2.Mnemonic then
        n = num_of_arguments(ins1)
        score += IDENTICAL_MNEMONIC_SCORE
        for $i = 0; i < n; ++i$ do
            if operand(ins1)[i] == operand(ins2)[i] then
                if type(operand(ins1)[i]) == CONSTANTS then
                    score += IDENTICAL_CONSTANT_SCORE
                else
                    score += IDENTICAL_OPERAND_SCORE
                end
            end
        end
    else
        score = 0
    end
    return score
end
```

Instead of comparing original instructions, we choose to compare the normalized instructions. The first advantage is more resistance to register reassignment, which is very common in compiler optimization. Second, we want to do a fuzzy matching. This is different from what David and Yahav did in [29], where they use exact matching when comparing operands. Besides, we allow partial matching. For example, we give a score of 5 to instruction pair `cmp [eax],0` and `cmp ebx, 0`, although they are two types of instructions. The first instruction is comparing an immediate value with a memory reference while the second with an register. The reason for us to allow partial matching is because even for the same variable, compilers have the freedom to represent it as a register variable or a memory variable. Allowing partial matching can tolerate these differences.

3.4 Basic Block Comparison

Inspired by the recent work in [29], we leverage the longest common subsequence (LCS) method of dynamic programming [28] to compare two basic blocks. The LCS problem is to find the longest subsequence which is common to both sequences. Suppose we have two strings, s_1 = “ABCAE” and s_2 = “BAE”, and we want to find their LCS. Dynamic programming can be applied to solve this problem efficiently. The general idea is to break down the problem into smaller and simpler problems until the answer becomes straightforward. Table 2 shows the memoization table when using dynamic programming to calculate the length of the LCS of these two strings.

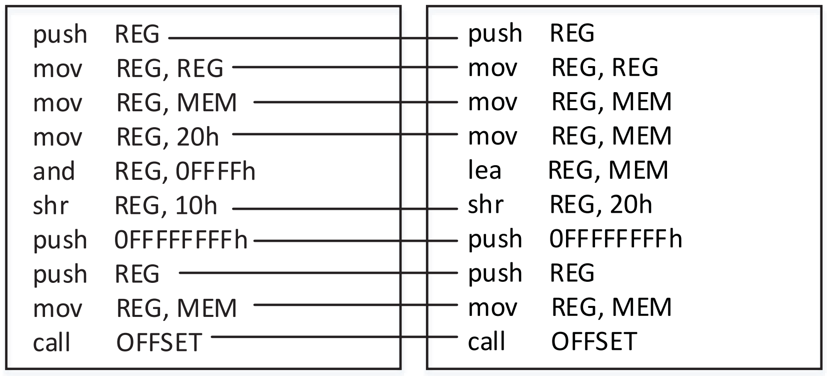
	s_1	A	B	C	A	E
s_2	0	0	0	0	0	0
B	0	$\leftarrow\uparrow$ 0	$\nwarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1	$\leftarrow$ 1
A	0	$\nwarrow$ 1	$\leftarrow\uparrow$ 1	$\leftarrow\uparrow$ 1	$\nwarrow$ 2	$\leftarrow$ 2
E	0	$\uparrow$ 1	$\leftarrow\uparrow$ 1	$\leftarrow\uparrow$ 1	$\uparrow$ 2	$\nwarrow$ 3

Table 2: The memoization table when calculating the LCS of two string, the highlighted cells show the backtrack path

Following the path highlighted in Table 2, we can obtain the longest common subsequence of these two strings, “BAE”, and its length is 3, as denoted in the last cell of the table.

Note that a basic block is also a sequence of assembly instructions. We then leverage the LCS to calculate the similarity score of two basic blocks. We consider every instruction as a letter and use the score strategy presented in Algorithm 1 to obtain the matching score. Notice that we do not draw any conclusion about whether these two basic blocks are identical or should be matched according to this score. Unlike the work in [29], we just use the similarity score as a guide for later use.

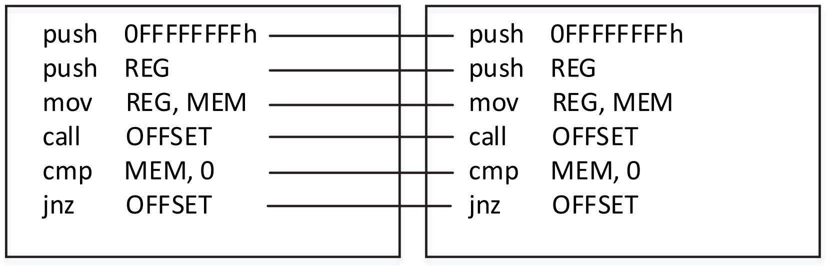
As shown in Algorithm 2, the output is the largest similarity score that these two basic blocks can achieve with respect to the score strategy we are using. By



```
push   REG                push   REG
mov    REG, REG           mov    REG, REG
mov    REG, MEM           mov    REG, MEM
mov    REG, 20h           mov    REG, MEM
and    REG, 0FFFFh        lea    REG, MEM
shr    REG, 10h           shr    REG, 20h
push   0FFFFFFFFh         push   0FFFFFFFFh
push   REG                push   REG
mov    REG, MEM           mov    REG, MEM
call   OFFSET             call   OFFSET
```

Figure 13: Example of instruction alignment and the lines represent the mapping of instructions that gives the highest similarity score

backtracking the memoization table, we can also obtain the mapping of instructions between this two basic blocks. Some literatures such as [29] also denote this process of leveraging dynamic programming to obtain the mapping, as “alignment”. After this “alignment”, instructions that cannot be matched can be jumped over. This jumping over instructions is our fuzzy matching at the basic block level. However, for now this mapping is of no interest to us, as we only need the maximum similarity score. Note that there may be different mappings that give us the same maximum score, however, the maximum score is unique. In our algorithm, it is always in the last cell of the memoization table.



```
push   0FFFFFFFFh         push   0FFFFFFFFh
push   REG                push   REG
mov    REG, MEM           mov    REG, MEM
call   OFFSET             call   OFFSET
cmp    MEM, 0             cmp    MEM, 0
jnz    OFFSET             jnz    OFFSET
```

Figure 14: Example of comparing a basic block with itself and the lines represent the mapping of instructions that gives the highest similarity score

A special case is to use Algorithm 2 to compare a basic block with itself. No doubt that the highest score will be achieved only when every instruction is mapped to itself as Figure 14 shows. We define that score as the "self" score of that basic block. Intuitively, this score can be used to measure the information that a basic block carries. A large basic block results in a high self score.

Algorithm 2: Calculate the similarity score of two basic blocks

```
Input: Two basic blocks $BB1, BB2$
Output: The similarity score of two basic blocks
/* M: the memoization table                                  */
Algorithm CompBBs($BB1, BB2$)
    M = InitTable($|BB1| + 1, |BB2| + 1$)
    for $i = 1; i <= |BB1|; ++i$ do
        for $j = 1; j <= |BB2|; ++j$ do
            $M[i, j] = Max($
            $CompIns(BB1[i], BB2[j]) + M[i-1, j-1],$
            $M[i-1, j],$
            $M[i, j-1])$
        end
    end
    return $M[|BB1|, |BB2|]$
end
```

3.5 Longest Path Generation

We have explained how to compare two basic blocks. For every basic block pair, we can obtain a similarity score. The larger the score, the more similar these two basic blocks are. However, this score is derived from the assembly code only, and is thus not sufficient. For example, for one target basic block, we might find multiple basic blocks that have the same similarity score with it. Even worse, we may end up matching it with a wrong basic block simply because its assembly code is more similar to the target by chance.

Inspired by the recent work in [56], we realize that path in the CFG is a robust feature, since path can record every selection the execution flow took when a branch is encountered, and one path represents one complete particular execution. Notice that the functionality of one path is spread across consisting nodes (basic blocks). If we succeed in finding two paths that are equivalent in terms of functionality, it would be trivial to further match their nodes. Again, we can treat the problem of finding matching nodes as an alignment problem where dynamic programming can be applied. Intuitively, one short path does not carry as much information as a long path. Besides, the longer the path, the more matching nodes we could obtain by aligning it with its matching path, which improves both the accuracy and efficiency of neighborhood exploration process (Section 3.7). Thus, we choose the longest path. We use depth first search to traverse the CFG, and then choose the path with the largest number of nodes.

3.6 Path Exploration

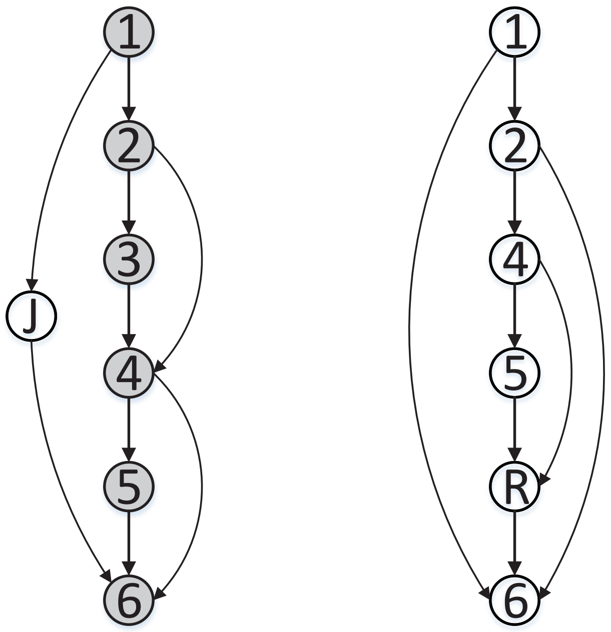
After we obtained the longest path of the target function, the next step is to explore the reference function, to try to find the best match of that path in the reference function. We adopt the approach in [56] to do the exploration. In [56] Luo *et al.* used a breadth-first search combined with dynamic programming to compute the highest score of longest common subsequence of semantically equivalent basic blocks. In our case, we leveraged their algorithm to find the corresponding path which has the largest similarity score based on Algorithm 2.

The algorithm for path exploration is similar to the common dynamic programming for computing the LCS of two strings. Since a path is also a sequence of basic blocks, we can treat every basic block as a letter and use the Algorithm 2 as our score strategy. However, there are two differences. First, the length of a string is constant, thus when computing the LCS of two strings the length of the memoization table is also fixed. In path exploration, however, we do not know the length of the memoization table in advance, so we set the initial length to one (Line 2 in Algorithm 3) and add more rows on the run (Line 9 in Algorithm 3). Second, the letters in a given string are sequential; every previous letter has at most one letter following it while a node in a CFG may have multiple successors. That is why we need to combine breadth-first search with the original dynamic programming.

We modified the algorithm in [56] to fit our needs. Given a longest path P from the target function and the CFG G of the reference function, we always start from the head node of G (Line 5 in Algorithm 3). At the beginning of each iteration, we pop out a node from the working queue Q as the current node (Line 7 in Algorithm 3). Then we add a new row to the memoization table δ and update the table correspondingly using function LCS (Line 10 in Algorithm 3). It is worth noting that when comparing the current node with every node in path P, we require them to have the same in-degree and out-degree to be matched (Line 22 in Algorithm 3). Otherwise we do not allow them to match by giving them a score of 0 (Line 25 in Algorithm 3). The motivation is that we want to quickly match the "skeleton" of the CFG first. If we failed to match some nodes whose in-degree or out-degree have been changed, we can leave them to the next step, neighborhood exploration. Also note that because of the complexity of the CFG, there might be multiple paths that can lead the execution flow to a certain node. To improve the efficiency, it is important to reduce the search space and prune the unprofitable path. To this end, we use an array σ to store the largest similarity score that we have achieved so far for each node. Every time after updating the table δ for certain node, we continue to compare the obtained new score with the largest score stored in σ (Line 11 in Algorithm 3). If the new score is larger, we then update σ and insert every successor of this node to our working queue Q. Otherwise we do not further explore its successors. The algorithm terminates after Q is empty. The output is the memoization table δ.

Algorithm 3: Path Exploration

```
Input: $P$: the longest path from the target function, $G$: the CFG of the
       reference function
Output: $\delta$: The memoization table
/* $\sigma$: the array that stores the largest LCS score for every node
   in $G$                                                              */
1  Function PathExploration($P$,$G$)
2      $\delta$ = InitTable(1, $|P| + 1$)
3      $\sigma$ = InitArray($|G|$)
4      $Q$ = InitQueue()
5      $Q$.pushback($G_1$) //always start from the head node
6      while $Q$ is not empty do
7          $currNode$ = $Q$.front()
8          $Q$.pop_front()
9          $\delta$.AddNewRow() //always add a new row to $\delta$
10         LCS($currNode$,$P$) //compare $currNode$ with every node in $P$ and
           update the table $\delta$
11         if $\sigma(currNode) < \delta(currNode, |P|)$ then
12             $\sigma(currNode) = \delta(currNode, |P|)$
13             for each successor $s$ of $currNode$ do
14                 $Q$.pushback($s$)
15             end
16         end
17     end
18     return $\delta$
19 end
20 Function LCS($u$,$P$)
21     for each node $v$ of $P$ do
22         if $SameDegree(u,v)$ then
23             $sim = CompBB(u, v)$
24         else
25             $sim = 0$
26         end
27         $\delta(u, v) = Max($
28         $\delta(parent(u), parent(v)) + sim,$
29         $\delta(parent(u), v),$
30         $\delta(u, parent(v)))$
31     end
32 end
```



(a) The CFGs of two versions of the same functions and the grey nodes represent the longest path in the target CFG

	v	1	2	3	4	5	6
u	0	0	0	0	0	0	0
1	0	↖ 1	← 1	← 1	← 1	← 1	← 1
6	0	↑ 1	↑ 1	↑ 1	↑ 1	↑ 1	↖ 2
2	0	↑ 1	↖ 2	← 2	← 2	← 2	← 2
6	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↖ 3
4	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2
5	0	↑ 1	↑ 2	↑ 2	↑ 2	↖ 3	← 3
R	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↑ 2
R	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 2	↖ 3
6	0	↑ 1	↑ 2	↑ 2	↑ 2	↑ 3	↖ 4

(b) The memoization table

Figure 15: An example of path exploration for two CFGs

Figure 15 presents an example. Figure 15a shows two simplified CFGs of two functions from open source projects; the grey nodes denote the longest path we found in the target CFG. These two functions are from the same source code. However due to the noise introduced by the compiler, their structures are not isomorphism. Basic block J in the target function consists of one "JMP" instruction, directing the execution flow to the tail block "6". Basic block 3 in the target CFG modifies the value of a local variable in the stack. As can be seen from Figure 15a, basic block 3 does not have a corresponding basic block in the reference CFG because the compiler used the register "ESI" to represent this variable in the reference function. Moreover, the reference CFG has one more basic block R, that restores the original value of "ESI", and then directs the execution flow to the tail.

To do the path exploration, we first initialize the memoization table δ and array σ. Then we insert the head node 1 of the candidate CFG to the working queue Q. We compare node 1 with path P using the function LCS in Algorithm 3 and update the memoization table correspondingly. Notice that here for the purpose of simplicity, we assume that the matching score is either 1 or 0, while a true match has a score of 1, otherwise 0. Since node 1 has two successors, node 6 and 2. We insert them into Q and continue the exploration. Assume we visit node 6 first, then node 2. Node 6 has no successor, we then update the table δ for node 6, and continue to work on node 2. Node 2 has two successors, node 6 and node 4. We also insert them into our working queue. We first work on node 6. Note that this is the second time we insert node 6 into Q. The first time its parent node is 1, and the corresponding partial path is "`1`$\rightarrow$`6`", this time its parent node is 2 and the partial path is "`1`$\rightarrow$`2`$\rightarrow$`6`". We allow the same node to be inserted into Q as long as they represent different execution paths. Node 6 has no successor. After we finish comparing node 6 with every node in path P, the working queue Q has only one element: node 4. We then work on node 4. It is worth noting that although node 4 in the reference CFG has a corresponding node

(node 4) in the path P, the in-degrees of these two nodes are different. Thus, we give them a matching score of 0. Then we put the successors of node 4 into Q. Now Q has two elements, node 5, and node R (parent is node 4). We visit node 5 first, and put its successor node R into Q. Now Q has two elements, node R from node 4 (partial path "`1`$\to$`2`$\to$`4`$\to$`R`") and node R from node 5 (partial path "`1`$\to$`2`$\to$`4`$\to$`5`$\to$`R`"). Both elements will lead us to node 6, but with different LCS scores. The one from node 4 (complete path "`1`$\to$`2`$\to$`4`$\to$`R`$\to$`6`") will have a final score of 3 while the one from node 5 (complete path "`1`$\to$`2`$\to$`4`$\to$`5`$\to$`R`$\to$`6`") gives us a score of 4.

We can then backtrack the memoization table δ to get the corresponding path that has the largest sum of similarity score with the target longest path. However, during our experiments we found that considering only the sum of the similarity score may sometimes give undesirable results. We might wrongly match the target path with a long path in the reference CFG. So we decided to normalize the similarity score by taking the target and the found path into consideration. Recall that a path is a sequence of basic blocks, and the self score of one basic block b can be calculated as $\mathit{CompBB}(b, b)$ using Algorithm 2. Then the self score of a path is the sum of self scores of all the consisting basic blocks. We then normalize the score between the target path P and the found path P_f using the following equation:

$$\mathit{NormScore}(P, P_f) = \frac{\mathit{LCSScore}}{\mathit{Score}(P) + \mathit{Score}(P_f)}$$

where the $\mathit{LCSScore}$ is the score obtained from the memoization table δ and $\mathit{Score}()$ is a function that returns the self score of the given path.

We then choose the path with the highest normalized score. By backtracking the memoization table δ, we can obtain a mapping of basic blocks. In the example shown in Figure 15 we can obtain 4 matching basic block pairs: basic block 1 with 1, 2 with 2, 5 with 5 and 6 with 6 in the target and reference, respectively.

3.7 Neighborhood Exploration

While we can continue to extract more paths from the target function and match them in the reference function, this is not efficient. First, the path exploration process takes time. Besides, when we explore certain target path in the reference function, some of the basic blocks may have already been matched in previous paths and we cannot gain much by rematching them. Inspired by the work in [67], we decided to use a greedy, localized fuzzy matching approach to extend the existing mapping. Because we already have all the mappings from path exploration of the longest path, there is a high chance that we can find the correct basic block mapping between two functions.

We first put every matching basic block pair obtained from path exploration into a priority queue based on their similarity score. Then we choose the pair on the top, namely the pair with the largest similarity score as our starting point to initialize the search. We then explore the neighbors of the chosen basic block pair. Note that for every basic block pair in the queue, the two basic blocks have the same in-degree or out-degree. We first consider the successors of these two basic blocks if they have the same out-degree. If they both have only one successor, then we match their successors directly, unless it is inconsistent with the mapping we already have. If they both have more than one successor, then we leverage the Hungarian algorithm [63] to find the best mapping between the two sets of successors that maximize the sum of the similarity score. Similarly, if the found mapping is inconsistent with the mapping we already have, we discard the corresponding match but continue to check other successors. We then do the same to their predecessors if they have the same in-degree.

It is important to note that for these found mapping pairs, the corresponding basic blocks in the pair do not necessarily have the same in-degree or out-degree. If they have the same in-degree, we put them into the priority queue but only explore their predecessors later, when they become the element with the highest priority (similarity score) in the queue. If they have the same out-degree, we explore their successors. If neither their in-degree nor out-degree is the same, we still allow these two basic blocks to be matched, however, we do not put them into the priority queue. In other words, we do not explore their neighborhood, because the likelihood of them being a correct match is relatively lower. By doing this, we achieve a fuzzy matching between the basic blocks of two functions. At the same time, if we mismatched a pair of basic blocks, we can make sure that the error would not propagate as we require the same in-degree or out-degree when exploring the neighborhood. On the other hand, for basic blocks that are correctly matched, we could explore their neighborhood in two directions efficiently.

We continue to do this until the priority queue is empty, i.e., until there is no more neighbors to be explored, or all the neighbors have different in-degree and out-degree and can not be further explored. We then leverage the obtained matching basic block pairs to calculate the similarity between the target function and the reference function.

An assembly function can be looked at as a set of basic blocks, we then calculate the self score of a function by adding the self scores of all the consisting basic blocks. Given two functions, f and g, suppose γ is the set of all the matching basic block pairs we obtained during path exploration and neighborhood exploration, the similarity of these two functions can be calculated as follows:

$$Similarity(f, g) = \frac{2 \sum_{\forall (u,v) \in \gamma} CompBB(u, v)}{Score(f) + Score(g)}$$

where u, v are basic blocks, $u \in f$, $v \in g$ and $Score()$ is a function that returns the self score of the given function.

3.8 Filtering

We have introduced how to pairwise compare two functions. However, we still need to address the scalability problem, especially when dealing with large data sets. Suppose we have a function repository consisting of one million functions, to find similar functions to a given target function, we have to compare the target function with every function in the repository and rank the results. This is not efficient as a large number of functions are not similar to the target and should not be compared.

To this end, we adopt a heuristic approach to prune the search space by excluding functions that are not likely to be matched. We designed two filters, based on the number of basic blocks and function fingerprint similarity threshold, respectively.

Filtering By Number of Basic Blocks

The reason to filter by number of basic blocks is straightforward. It is very unlikely that a function with only one basic block can be matched to another function with one hundred basic blocks. Thus we set a number threshold. If we require two CFGs to be exactly the same, namely isomorphic, then the basic block numbers should also be the same. Since BinSequence performs a fuzzy matching, which allows two structurally-different functions to be matched, the numbers of basic block should be allowed to be different. So we set up a threshold. This threshold should not be too small, as we may rule out the correct match. On the other hand, the threshold should not be too large. Otherwise we cannot save much time as not many functions can be ruled out.

If the CFGs of two functions are isomorphic, they will have the same number of basic blocks. Even when they were slightly changed due to noise introduced by the compiler, the numbers are still very close. One thing that could change the basic block number is function inlining, where the assembly code of a small function was directly inlined in another function, to avoid the calling overhead. However, inlining only happens when the size of function is relatively small. So it will not significantly change the number of basic blocks.

Assume the threshold is γ, given a target function f, those functions whose sizes are between $|f| - \gamma$ and $|f| + \gamma$ will pass this filter.

Filtering By Fingerprint Similarity

The next filter is based on the syntactic property of the code. For every function, we use its normalized instruction set as its fingerprint. More specifically, we use the same technique as introduced in Section 3.2 to normalize all the instructions inside a function, to get the normalized instruction set. Given a target function, we then calculate the Jaccard similarity (index) between the fingerprints of the target and every function in the repository. If the Jaccard similarity is above a certain threshold, we then continue to compare the function against the target. Otherwise we simply discard it.

In order to avoid pairwise comparison of fingerprints, we leveraged minhashing [18] and the banding technique [69]. Minhashing is a technique of using k different hash functions to generate the minhash signature. The banding technique divides the minhash signature into b bands of r rows each. Given a target function, we first generate its fingerprint and the minhash signature of its fingerprint. We divide its minhash signature into b bands of r rows, each. Then the candidate set should be all the functions whose minhash signatures agree in all the rows of at least one band with the signature of the target function. More generally, if we choose n hash functions, b bands, r rows and $n = br$, the Jaccard similarity threshold t imposed by this banding technique is approximately $1/b^{1/r}$ [69].

In general, similar to the filter using number of basic blocks, this filter is lossy as well. Some true matches may have significantly different normalized instruction set, and consequently, fail to pass this filter. To address this problem, in our implementation, we choose b and r so that $t = 1/b^{1/r}$ equals to a relatively low value, e.g., 0.65, so that those functions that are true matches, but with low Jaccard similarity could pass this filter and remain in the candidate set. After using these techniques, to root out all the functions whose Jaccard similarity is above certain threshold, we only need to first choose b and r so that the desired threshold is imposed by the banding technique. and then select all the functions whose minhash signatures agree in all the row of at least one band with the signature of the target function, which can be achieved by one time lookup in the database.

3.9 Summary

We have presented the detailed design of our approach. Given a target function and a repository of thousands or millions of functions, we first use the filtering process to get a subset (candidate set) of promising functions from the repository that might be similar to the target. The filtering process consists of two filters. The first filter is based on the number of basic blocks, while the second is based on fingerprint similarity. After the candidate set has been obtained, we then compare the target function with each function in the candidate set using a fuzzy matching approach. The fuzzy matching approach operates at three different levels: instruction level, basic block level and control flow structure level. More specifically, we first generate the longest path of the target function. Then we explore the reference function in the candidate set to find the corresponding matching path, from which we can obtain the initial mapping of basic blocks. We then improve the mapping through neighborhood exploration in both the target and reference functions. The output is the mapping of basic blocks and the similarity score of these two functions. Finally, we rank the candidate functions based on their similarity scores.

Chapter 4

Evaluation

We conducted extensive experiments to evaluate BinSequence in terms of accuracy, performance and scalability. We also performed several experiments on practical scenarios to demonstrate the effectiveness and efficiency of BinSequence when applied to real-world use cases. All experiments were performed on a PC with an Intel Xeon E31220 Quad-Core processor, 16 GB of RAM running Microsoft Windows 7 64-bit.

4.1 Function Reuse Detection

The first experiment is function reuse detection from a large repository. We first try to perform function reuse detection between two versions of the same binary. In this experiment, four different versions of zlib libraries [15], namely 1.2.5 through 1.2.8 were used. Since zlib is a well maintained library, we assumed functions with identical function (symbolic) name across different versions should have the same or similar functionality, and thus, should be matched. We also introduced one group of noise functions, which are all the functions of 1,701 system dynamic library files obtained from Microsoft Windows operating system. The total size of these files is around 1 GB and the total number of functions is 2,055,584. It took BinSequence 14 hours and 52 minutes to import all these functions into the repository.

Every time we use the previous version of zlib to match the next version of zlib. For example, we first use zlib 1.2.5 as our target set, and put all the functions of its successive version zlib 1.2.6 together with the two million noise functions into the database. Then for every function (with at least four basic blocks) in zlib 1.2.5, we use BinSequence to search for it. Only when the corresponding function in zlib 1.2.6 is ranked the first, which means it has the highest similarity, we consider it as a correct match. Otherwise we consider it as wrongly matched. We then do the same thing to other versions of zlib.

For all the tests, we used three different fingerprint similarity thresholds: 0.6, 0.65 and 0.7. Using the techniques explained in Section 3.8, those functions whose fingerprint similarity below these thresholds would be ruled out. Intuitively as we increase the fingerprint similarity threshold, the number of functions that could pass this filter would decrease. So we choose three different values to thoroughly study its effect. The threshold for the number of basic blocks is set to 35 in this experiment. If we want to tolerate more "noise", we can also set this threshold to a larger value.

Table 3a shows the result. Recall that for every target function, we use two filters to obtain a small candidate set from the whole database. The column "Candidate Size" is the sum of the size of the candidate sets for every target function. Intuitively, as we increase the similarity threshold, we end up with a smaller candidate set. As a result, the processing time decreases. We expected that as we increase the fingerprint threshold, the overall accuracy would drop (like zlib 1.2.5 in the table), or at the best would stay the same (like zlib 1.2.6 in the table) because we would get a smaller candidate set and the true match could have been ruled out. The zlib 1.2.7 was a surprise. As we increased the fingerprint threshold from 0.6 to 0.7, the overall accuracy increased from 96.52% to 98.26%. We looked into the reason. When the fingerprint threshold is 0.6, there were two functions, whose true matches did not have the largest similarity; instead, two other functions that happened to have similar code

Version	Fingerprint Threshold	Overall Accuracy	Candidate Size	Time (per function)
1.2.5	0.6	96.26%	12346	2.806s
	0.65	94.39%	2727	1.468s
	0.7	91.59%	1911	0.897s
1.2.6	0.6	100%	16315	2.927s
	0.65	100%	2848	1.558s
	0.7	100%	1989	0.913s
1.2.7	0.6	96.52%	16312	2.884s
	0.65	97.39%	2847	1.572s
	0.7	98.26%	1988	0.918s

(a) Function reuse detection between different versions of zlib

Version	Fingerprint Threshold	Overall Accuracy	Candidate Size	Time (per function)
1.2.8	0.6	92.5%	3526	2.204s
	0.65	92.5%	751	1.258s
	0.7	92.5%	242	0.95s

(b) Function reuse detection between zlib and libpng

Table 3: Results for function reuse detection

and structure were ranked first. The true matches were ranked #2. As we increased the fingerprint similarity threshold from 0.6 to 0.7, these two functions were ruled out by the filter; as a result, those true matches become the ones with the highest similarity. This also suggest that though our filters are in general lossy, however do not necessarily always decrease the accuracy.

We also conducted function reuse detection between two different binaries: zlib and libpng [11]. Libpng is a library for processing PNG image format files and it is dependent on zlib library. As a result, part of the functions from zlib library are reused by libpng. We first compiled zlib 1.2.8 and libpng 1.6.17 with the debugging information attached. By manually checking both libraries, we identified 40 functions that were user functions in zlib and were reused in libpng. We then used these 40 functions (with at least four basic blocks) in zlib as our target functions, and searched for them in the repository. If the corresponding function in libpng is ranked first, we consider it as a correct match. As shown in Table 3b, we correctly identified 37 reused functions for all the three different fingerprint thresholds, and the overall accuracy was consistently 92.5%.

4.2 Patch Analysis

The next experiment is to use BinSequence to recover vulnerability information. Nowadays as a result of the development of vulnerability mining techniques, more vulnerabilities are being discovered everyday. After a vulnerability is reported to the software vendor, they would release a security patch to fix it often without revealing the details of the vulnerability or the part of code they have modified to the public. By comparing the patched and unpatched version of the binaries, reverse engineers can analyze and understand the vulnerability and the patch within hours. This kind of technique is especially useful for Microsoft's binaries as they release the patch regularly and the patched vulnerability are concentrated in small areas in the binary [66].

We take a recently patched vulnerability, namely `MS15-034` [12] as a case to study. There is a vulnerability in `HTTP.sys`. When an attacker sends a specially crafted HTTP request to an affected system, the HTTP protocol stack may parse it improperly. As a result, the attacker may execute arbitrary code. Microsoft released a patch MS15-034 to address this problem. In order to reveal the information of the vulnerability and the patch, we used BinSequence to compare the unpatched and patched version of `HTTP.sys`. Since this is to find out which functions have been patched, we only report functions whose similarity is not 1 after being patched, as similarity 1 means the function remains identical (after normalization) in the patched version.

In total, BinSequence identified 11 functions, whose similarity is not 1 between the patched and unpatched version. We manually checked these 11 function pairs and found out that 6 functions were actually the same, but were disassembled differently by IDA Pro. Table 4 lists the remaining 5 functions.

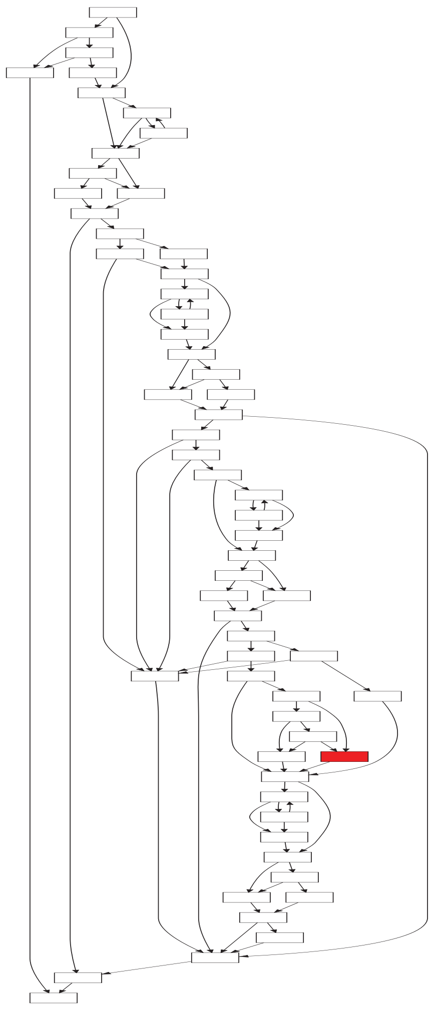
Function	Similarity
`UlpParseRange`	0.971783
`UlpBuildSingleRangeMdlChainFromSlices`	0.915530
`UlpBuildMultiRangeMdlChainFromSlices`	0.849870
`UlpDuplicateChunkRange`	0.804167
`UlAdjustRangesToContentSize`	0.501853

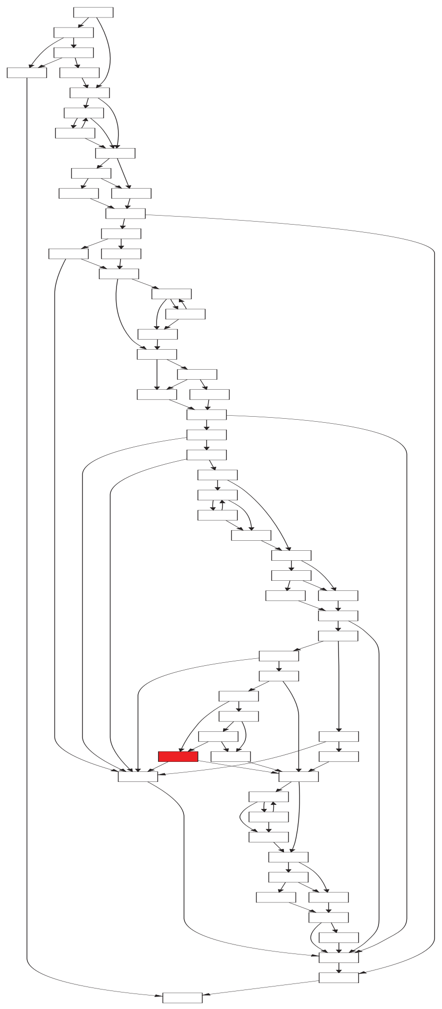
Table 4: Patched functions

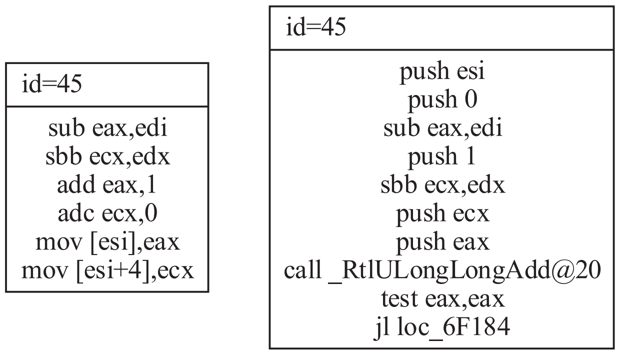
According to the analysis in [13], among these five function pairs three of them are related to the vulnerability, namely `UlpParseRange`, `UlpDuplicateChunkRange` and `UlAdjustRangesToContentSize`. We first take a look at the `UlpParseRange` function. As shown in Figure 16a and 16b, both the pathched and unpatched versions have 60 basic blocks. BinSequence successfully matched all the basic blocks. Among all these pairs, 59 pairs have a similarity of 1, which means they remain the same after being patched (after normalization). The only basic block pair highlighted in red in Figure 16a and 16b shows where the patch took place, basic block number 45.

Figure 16c depicts the basic block 45 before and after the patch. We can clearly see that the patched version was calling a function `RtlUlongLongAdd` while the unpatched version does not. We can infer that the original function might contain an integer overflow vulnerability. The patched version invokes the `RtlULongLongAdd` to fix it. Moreover, we can see that their out-degrees of these two basic blocks have been changed. The out-degree of the unpatched is 1 while the patched is 2. Despite this structure change, our fuzzy structure matching approach still succeed in matching these two basic blocks.

We continue to look at the `UlpDuplicateChunkRange` function. As depicted in Figure 17a and 17b, the unpatched version has 31 basic blocks while the patched has 37 basic blocks. BinSequence successfully matched all the 31 basic blocks in the


(a) The *UlpParseRange* function before patch


(b) The *UlpParseRange* function after patch

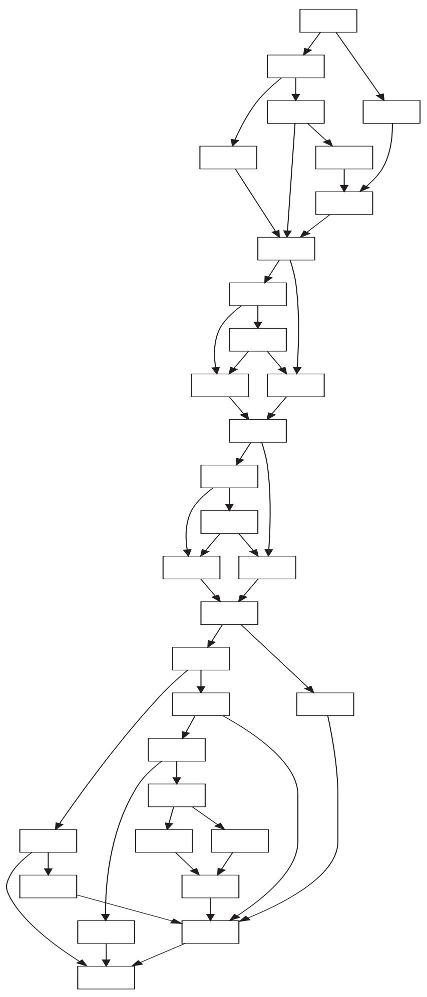


```
id=45

sub eax,edi
sbb ecx,edx
add eax,1
adc ecx,0
mov [esi],eax
mov [esi+4],ecx
```

```
id=45

push esi
push 0
sub eax,edi
push 1
sbb ecx,edx
push ecx
push eax
call _RtlULongLongAdd@20
test eax,eax
jl loc_6F184
```

(c) The basic block 45 before and after patched, the out-degree is 1 and 2, correspondingly

Figure 16: The *UlpParseRange* function


(a) The *UlpDuplicateChunkRange* function before patch

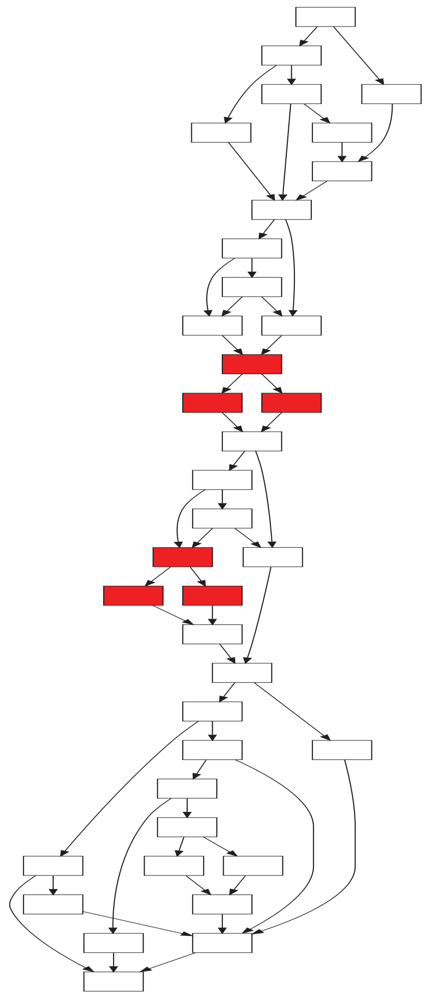
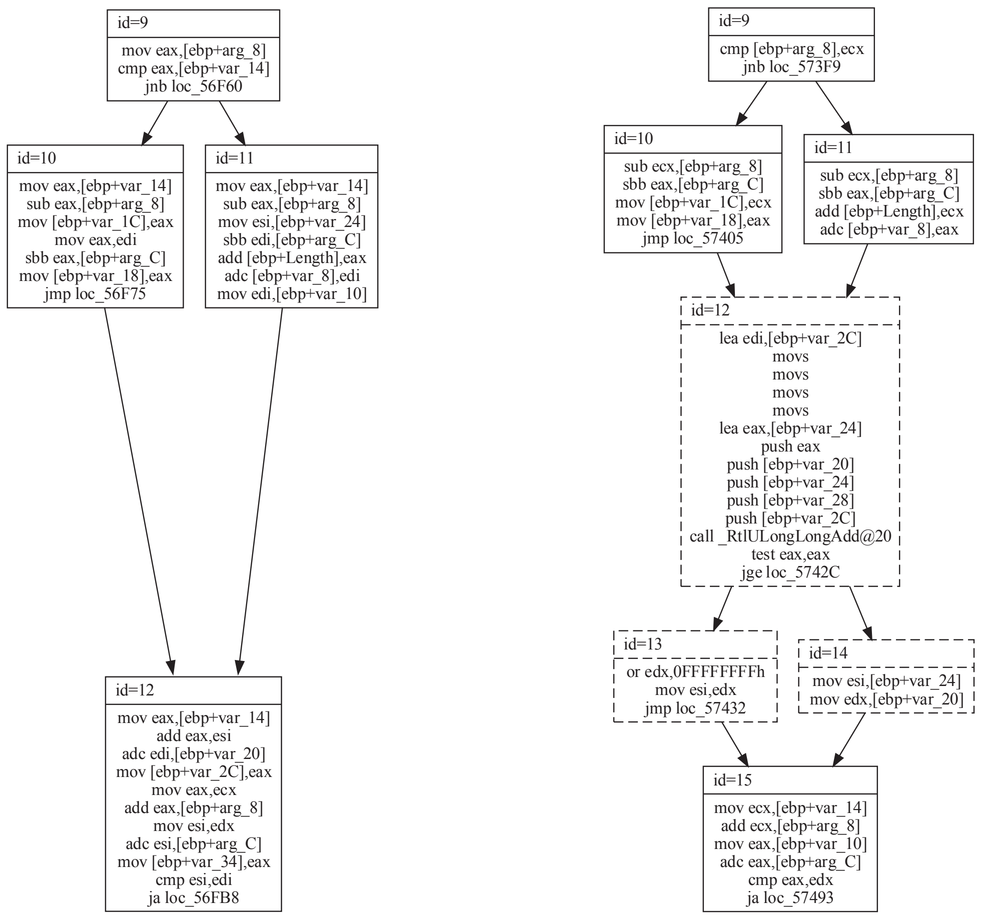

(b) The *UlpDuplicateChunkRange* function after patch

Figure 17: The *UlpDuplicateChunkRange* function

unpatched version to their counterparts in the patched version. Six basic blocks in the patched version are not matched, thus, we can infer that they are inserted to patch the vulnerability. Figure 18 shows the detail of the first three inserted basic blocks. Basic block 9, 10, 11, 12 in the unpatched function are matched to basic block 9, 10, 11, 15 in the patched function, respectively. Basic block 12, 13, 14 in the patched are newly inserted. We can see that the inserted basic block 12 invokes the `RtlULongLongAdd` function as well and makes one conditional jump based on the return value. Our fuzzy matching approach successfully jumped over these inserted basic blocks and matched the rest of control flow graphs.



```
id=9
mov eax,[ebp+arg_8]
cmp eax,[ebp+var_14]
jnb loc_56F60

id=10
mov eax,[ebp+var_14]
sub eax,[ebp+arg_8]
mov [ebp+var_1C],eax
mov eax,edi
sbb eax,[ebp+arg_C]
mov [ebp+var_18],eax
jmp loc_56F75

id=11
mov eax,[ebp+var_14]
sub eax,[ebp+arg_8]
mov esi,[ebp+var_24]
sbb edi,[ebp+arg_C]
add [ebp+Length],eax
adc [ebp+var_8],edi
mov edi,[ebp+var_10]

id=12
mov eax,[ebp+var_14]
add eax,esi
adc edi,[ebp+var_20]
mov [ebp+var_2C],eax
mov eax,ecx
add eax,[ebp+arg_8]
mov esi,edx
adc esi,[ebp+arg_C]
mov [ebp+var_34],eax
cmp esi,edi
ja loc_56FB8
```

```
id=9
cmp [ebp+arg_8],ecx
jnb loc_573F9

id=10
sub ecx,[ebp+arg_8]
sbb eax,[ebp+arg_C]
mov [ebp+var_1C],ecx
mov [ebp+var_18],eax
jmp loc_57405

id=11
sub ecx,[ebp+arg_8]
sbb eax,[ebp+arg_C]
add [ebp+Length],ecx
adc [ebp+var_8],eax

id=12
lea edi,[ebp+var_2C]
movs
movs
movs
movs
lea eax,[ebp+var_24]
push eax
push [ebp+var_20]
push [ebp+var_24]
push [ebp+var_28]
push [ebp+var_2C]
call _RtlULongLongAdd@20
test eax,eax
jge loc_5742C

id=13
or edx,0FFFFFFFFh
mov esi,edx
jmp loc_57432

id=14
mov esi,[ebp+var_24]
mov edx,[ebp+var_20]

id=15
mov ecx,[ebp+var_14]
add ecx,[ebp+arg_8]
mov eax,[ebp+var_10]
adc eax,[ebp+arg_C]
cmp eax,edx
ja loc_57493
```

(a) Before patch (b) After patch

Figure 18: One of the patched parts of the *UlpDuplicateChunkRange* function

4.3 Malware Analysis

Our next experiment is conducted on two well known malware, Citadel and Zeus. We know that the Citedel is derived from Zeus [62]. We also know that Zeus uses RC4 stream cipher function, and Citadel reused this function with modification [68]. Given the RC4 function in Zeus, our intention is to use BinSequence to identify the reused RC4 function in Citadel.

We first disassembled Zeus using IDA Pro and extracted the RC4 function. We then used it as our target function. We also disassembled Citadel and then compared the target function with every function in Citadel, and ranked the results according to the similarity score. Table 5 shows the top 3 functions that have the largest similarity.

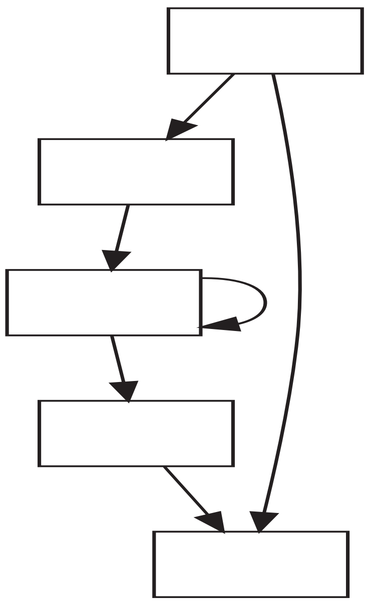
Function	Similarity
`sub_42E92D`	0.689474
`sub_432877`	0.429091
`sub_430829`	0.423913

Table 5: The result of searching RC4 function in Citadel

We manually checked the sub_42E92D function, and confirmed that this is the modified RC4 function in Citadel. In total IDA Pro identified 794 assembly function in Citadel. That is to say, we successfully identified the modified RC4 function from these 794 functions.

Since 794 functions is a relatively small data set, so we put the RC4 function in Citadel into those 2 million functions we used in Section 4.1 and redid the experiment. Still, BinSequence ranked the modified RC4 function as first, from a function repository with more than 4 million functions.

Figure 19a and Figure 19b show the RC4 function in Zeus and Citadel respectively. Clearly we can see these two CFGs are by no means isomorphic, yet BinSequence ranked the modified RC4 first with the highest similarity. Again, this result demonstrates that our fuzzy matching approach is effective and accurate.


(a) The RC4 function in Zeus

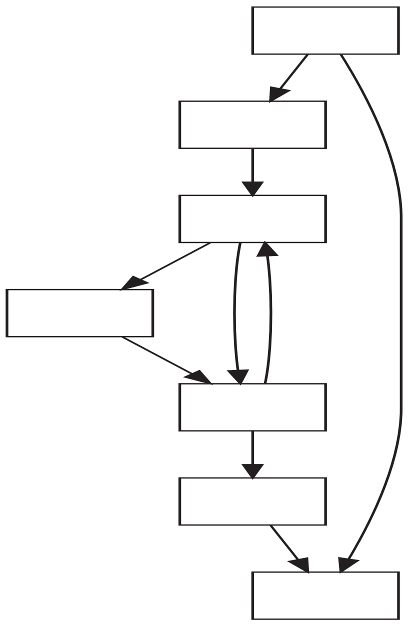

(b) The modified RC4 function in Citadel

Figure 19: The RC4 function

We also looked into the differences between the RC4 functions of these two malware. Unlike Zeus, the RC4 function in Citadel has a MD5 hashed login key [68]. At the beginning of the RC4 function of Citadel, the `strlen` function is invoked to calculate the length of this hashed login key. Following is a conditional jump instruction that direct the execution flow to two different branches based on the length of the login key. Other than that, an additional encryption step is performed using this hashed login key before generating the final encryption output. As a result, both the control flow structure and the basic blocks of the RC4 in Citadel have been changed compared to the RC4 function in Zeus. This is the reason why the similarity between these two RC4 functions is only 0.689474, which is relatively low. In fact, most of the functions in Zeus have counterparts in Citadel with a higher similarity. More specifically, for 373 (67%) functions in Zeus, Binsequence identified matches in Citadel with a similarity of 1.0, which means they are exactly identical (after normalization) and

513 (92.1%) functions with a similarity above 0.8. This also confirms that Citadel reused most of the Zeus's functions (functionality). Now when reverse engineering Citadel, the human analyst can focus on those new components and functionality, instead of reanalyzing these reused functions.

4.4 Bug Search

The next experiment is also a use case bug search. There is a heap-based buffer overflow in `resize_context_buffers` function in libvpx library and Firefox uses this library [5]. Our intention is to use the `resize_context_buffers` function in libvpx as our target function and identify the buggy function in the repository if there is any.

According to the vulnerability datasource [5], this bug only exists in Firefox before 40.0.0. So in the first experiment, we used Firefox 39.0.0 and 40.0.0 as the first contains this vulnerability and the second does not. We compiled Firefox by ourselves with debug symbols so that we can see the function name when disassembling Firefox. Table 6 shows the results. BinSequence successfully ranked the vulnerable `resize_context_buffers` function in Firefox 39.0.0 and its patched version in Firefox 40.0.0 as first.

Firefox	Function	Similarity	Rank
39.0.0	`resize_context_buffers`	1.000000	1/149,784
40.0.0	`resize_context_buffers`	0.709524	1/152,618

Table 6: Search results for Firefox 39.0.0 and 40.0.0

In total Firefox 39.0.0 and 40.0.0 have 302,402 functions. By setting the basic number number threshold to 35, and the fingerprint similarity threshold to 0.6, we end up with a candidate set with a size of 5. The overall running time was 0.468s.

The reason that `resize_context_buffers` function in Firefox 39.0.0 has a similarity of 1 with the target is that we used the same compiler and optimization level to compile both functions, as a result their code are highly similar. The `resize_context_buffers` function in Firefox 40.0.0 has been patched, so their similarity is no longer 1, but still it has the largest similarity with the target among these 152,618 functions.

We also downloaded different versions of Firefox from 33.0.0 to 40.0.0 from the official web site [8] directly. We only considered the main version and ignore those subversions like 33.0.1.

Firefox	Function	Similarity	Rank
40.0.0	`sub_116D3D02`	0.427699	1/161,932
39.0.0	`sub_1165C97B`	0.657224	1/159,589
38.0.0	`sub_1153BA02`	0.657224	1/155,299
37.0.0	`sub_1155BD63`	0.657224	1/151,416
36.0.0	`sub_115F7CB3`	0.657224	1/152,032
35.0.0	`sub_100CB36B`	0.268199	1/142,304
34.0.0	`sub_101800DA`	0.268199	1/138,329
33.0.0	`sub_108F3DA4`	0.141892	1/135,621

Table 7: Search results for different versions of Firefox

In total there are 1,196,522 functions in these 8 versions of Firefox, and it took 0.271 second for BinSequence to finish the whole comparison. As the Table 7 shows, BinSequence uniquely identified the equivalent buggy function in Firefox 36.0.0, 37.0.0, 38.0.0, 39.0.0. We manually checked the assembly and the source code and confirmed the found functions indeed are the buggy functions. For Firefox 33.0.0 through 35.0.0, BinSequence found three functions with a relatively low similarity. We found that these three versions of Firefox were using a different version of libvpx. As a result, the buggy function actually does not exist in these three versions. Still, BinSequence reported the ones with the highest similarity in the corresponding project.

4.5 Function Matching

In this experiment, we compare BinSequence with Diaphora [6], PatchDiff2 [14] and BinDiff [3] for function matching. Both Diaphora and PatchDiff2 are IDA Pro plugins for program diffing and BinDiff is the de facto standard commercial tool for comparing binary files. All these tools can compare two versions of the same binary and create a mapping of functions between them. It is worth noting that BinSequence is not confined to comparing two binaries. In this experiment, given a target function in one binary, we use BinSequence to compare the target with every function in the other binary and match the target to the function with the largest similarity score.

Throughout this experiment, we continue to use zlib 1.2.8. However, we compile it in release mode using two different compilers, namely MSVC 2010 and MSVC 2013. The reason of choosing these two compilers is to introduce certain "noise" into the code. We then use the functions in zlib 1.2.8 compiled by MSVC 2010 as out target set, and functions compiled by MSVC 2013 as the candidate set. For every non-empty function (with at least 4 instructions) in the target set, we use BinSequence to find the matching function (with the highest similarity) in the candidate set. In this experiment, all the function names of both binaries are stripped away. But we use the function names in the program debug database as the ground truth, to verify if the matching is correct.

Tool	Correctly Matched	Unmatched	Mismatched	Overall Accuracy
Diaphora [6]	105	10	29	72.92%
PatchDiff2 [14]	110	28	6	76.39%
BinDiff [3]	130	5	9	90.28%
BinSequence	135	0	9	93.75%

Table 8: Comparison with other tools

Table 8 shows the results. In total, our target set has 144 functions with more than 4 instructions. Diaphora correctly matched 105 functions. However, for 10 functions, Diaphora failed to match them and categorized them to "Unmatched" group. Moreover, Diaphora mismatched 29 functions. The overall accuracy for Diaphora is about 72.92%. PatchDiff2 correctly matched 110 function and mismatched 6 functions. For 28 functions, PatchDiff2 failed to find any match. Similarly, if BinDiff failed to match one function with another, BinDiff will classify it into "Unmatched". As shown in Table 8, there are 5 functions that BinDiff failed to match. However, Given one target function, BinSequence simply compares it with every function in the candidate set and match it to the one with the highest similarity. As a result, BinSequence has no "Unmatched" category.

We can see from Table 8 that BinSequence achieves the highest accuracy, 93.75%. The reason is that BinSequence is performing a fuzzy matching, which can better address the mutations introduced by different compilers. Also note that during the whole comparison process, BinSequence did not take the call graph into consideration. On the contrary, BinDiff would leverage call graph to match as many functions as possible. By considering the control flow graph alone, BinSequence achieved a higher accuracy than BinDiff. These results again, demonstrate the effectiveness of our fuzzy matching approach.

Chapter 5

Conclusion and Future Work

In this thesis we presented a fast, accurate and scalable binary code reuse detection system named BinSequence. Unlike previous works, we focus on fuzzy matching that operates at instruction level, basic block level and control flow structure level. To enable BinSequence on large data sets, we designed two filters to save analysis effort by ruling out functions that are not likely to be matched. Experiments were conducted on sheer volume of executables, and the results strongly suggest that BinSequence can achieve high quality function ranking.

We also applied BinSequence on many practical use case. By leveraging BinSequence on both patched and unpatched executables, we succeeded in revealing the vulnerability and the patch information. By performing function reuse detection, we managed to identify the reused RC4 function in two real-world malware, Zeus and Citadel. We also successfully identified the buggy function in various versions of Firefox. The comparison of BinSequence with three state of the art tools also suggests that our tool can achieve the best accuracy when compared with these tools. We believe that BinSequence can be of great help in many reverse engineering and security scenarios.

However, during experiments we also observed several limitations. We now briefly introduce them and propose some solutions as future directions.

- Function inlining: If a target function is inlined in another function, then our approach may not match these two functions. However, normally function inlining only happens to small functions. Consequently, their functionality are straightforward, and it does not really help reverse engineers much to search for them from a large function repository. On the other hand, if our target function inlined another small function, then still there is a high chance that BinSequence can match them since we are doing a fuzzy matching.

- Equivalent instructions: The compiler may use different instructions to accomplish the same functionality. For example, `mov eax, 0` and `xor eax, eax` have the same functionality but different mnemonics. However, they will be normalized to different instructions by our approach. Future versions of BinSequence may overcome this limitation by dividing instructions into different classes and let instructions in the same class to be matched.

- Control flow flattening: When comparing functions, we take the structure of their CFGs into consideration. So control flow flattening could pose a problem. In fact, during experiments, we encountered two versions of the same function, one with a size of 9 basic blocks while the other with only 1 basic block. For example, the compiler can use one condition move instruction like `cmovbe` to perform the job that originally requires two different branches in CFG. In that case it is hard for us to match these two functions. Future work may involve merging basic blocks, or spiliting one basic into multiple basic blocks, to achieve a better matching for this case.

Besides, for now BinSequence relies on the longest path to generate matching basic block pairs as starting points to initiate the neighborhood search. Future versions of BinSequence may take more paths into account, or leverage other different techniques to generate more starting points more efficiently.

Bibliography

[1] Anubis. `http://anubis.iseclab.org/`. Accessed: 2015-11-11.

[2] Bagle trojan. `https://en.wikipedia.org/wiki/Bagle_(computer_worm)`. Accessed: 2015-11-11.

[3] BinDiff. `http://www.zynamics.com/bindiff.html`. Accessed: 2015-11-11.

[4] Cosine distance. `https://en.wikipedia.org/wiki/Cosine_similarity`. Accessed: 2015-11-11.

[5] CVE-2015-4485. `http://www.cvedetails.com/cve/CVE-2015-4485/`. Accessed: 2015-11-11.

[6] Diaphora. `https://github.com/joxeankoret/diaphora`. Accessed: 2015-11-11.

[7] FCatalog. `http://www.xorpd.net/pages/fcatalog.html`. Accessed: 2015-11-11.

[8] Firefox. `https://www.mozilla.org/`. Accessed: 2015-11-11.

[9] IDA Pro. `https://www.hex-rays.com/products/ida/`. Accessed: 2015-11-11.

[10] Jaccard index. `https://en.wikipedia.org/wiki/Jaccard_index`. Accessed: 2015-11-11.

[11] Libpng library. `http://www.libpng.org/`. Accessed: 2015-11-11.

[12] MS15-034. `https://technet.microsoft.com/en-us/library/security/ms15-034.aspx`. Accessed: 2015-11-11.

[13] MS15-034 analysis and remote detection. `https://community.qualys.com/blogs/securitylabs/2015/04/20/ms15-034-analyze-and-remote-detection`. Accessed: 2015-11-11.

[14] PatchDiff2. `https://code.google.com/p/patchdiff2/`. Accessed: 2015-11-11.

[15] Zlib library. `http://www.zlib.net/`. Accessed: 2015-11-11.

[16] Mohamed Ibrahim Abouelhoda, Stefan Kurtz, and Enno Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

[17] Allan J Albrecht and John E Gaffney Jr. Software function, source lines of code, and development effort prediction: a software science validation. *Software Engineering, IEEE Transactions on*, (6):639–648, 1983.

[18] Alexandr Andoni and Piotr Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*, pages 459–468, 2006.

[19] László Babai and Eugene M Luks. Canonical labeling of graphs. In *Proceedings of the 15th annual ACM symposium on Theory of computing*, pages 171–183, 1983.

[20] Hamid Abdul Basit and Stan Jarzabek. Efficient token based clone detection with flexible tokenization. In *Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 513–516, 2007.

[21] Ira D Baxter, Andrew Yahin, Leonardo Moura, Marcelo Sant Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *Proceedings of the 14th International Conference on Software Maintenance*, pages 368–377, 1998.

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

[23] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[24] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Detecting self-mutating malware using control-flow graph matching. In *Detection of Intrusions and Malware & Vulnerability Assessment*, pages 129–143. Springer, 2006.

[25] Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Using code normalization for fighting self-mutating malware. In *Proceedings of the International Symposium on Secure Software Engineering*, pages 37–44, 2006.

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

[27] Luigi Pietro Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento. An improved algorithm for matching large graphs. In *3rd IAPR-TC15 workshop on graph-based representations in pattern recognition*, pages 149–159, 2001.

[28] Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009.

[29] Yaniv David and Eran Yahav. Tracelet-based code search in executables. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 349–360, 2014.

[30] Stéphane Ducasse, Matthias Rieger, and Serge Demeyer. A language independent approach for detecting duplicated code. In *Proceedings of the 15th International Conference on Software Maintenance*, pages 109–118, 1999.

[31] Thomas Dullien, Ero Carrera, Soeren-Meyer Eppler, and Sebastian Porst. Automated attacker correlation for malicious code. Technical report, DTIC Document, 2010.

[32] Thomas Dullien and Rolf Rolles. Graph-based comparison of executable objects (english version). *SSTIC*, 5:1–3, 2005.

[33] Eldad Eilam. *Reversing: secrets of reverse engineering.* John Wiley & Sons, 2011.

[34] Mohammad Reza Farhadi, Benjamin Fung, Philippe Charland, and Mourad Debbabi. Binclone: Detecting code clones in malware. In *Proceedings of the 8th International Conference on Software Security and Reliability*, pages 78–87, 2014.

[35] Halvar Flake. Structural comparison of executable objects. 2004.

[36] Debin Gao, Michael K Reiter, and Dawn Song. Binhunt: Automatically finding semantic differences in binary programs. In *Information and Communications Security*, pages 238–255. Springer, 2008.

[37] Yoann Guillot and Alexandre Gazet. Automatic binary deobfuscation. *Journal in computer virology*, 6(3):261–276, 2010.

[38] Dan Gusfield. *Algorithms on strings, trees and sequences: computer science and computational biology.* Cambridge university press, 1997.

[39] Sallie Henry and Dennis Kafura. Software structure metrics based on information flow. *Software Engineering, IEEE Transactions on*, (5):510–518, 1981.

[40] Jiyong Jang and David Brumley. Bitshred: Fast, scalable code reuse detection in binary code (cmu-cylab-10-006). *CyLab*, page 28, 2009.

[41] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th international conference on Software Engineering*, pages 96–105, 2007.

[42] J Howard Johnson. Substring matching for clone detection and change tracking. In *Proceedings of the 10th International Conference on Software Maintenance*, pages 120–126, 1994.

[43] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Ccfinder: a multi-linguistic token-based code clone detection system for large scale source code. *Software Engineering, IEEE Transactions on*, 28(7):654–670, 2002.

[44] Md Enamul Karim, Andrew Walenstein, Arun Lakhotia, and Laxmi Parida. Malware phylogeny generation using permutations of code. *Journal in Computer Virology*, 1(1-2):13–23, 2005.

[45] Richard M Karp. Combinatorics, complexity, and randomness. *Commun. ACM*, 29(2):97–109, 1986.

[46] Richard M Karp and Michael O Rabin. Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, 31(2):249–260, 1987.

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

[48] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[49] Raghavan Komondoor and Susan Horwitz. Using slicing to identify duplication in source code. In *Static Analysis*, pages 40–56. Springer, 2001.

[50] Kostas A Kontogiannis, Renator DeMori, Ettore Merlo, Michael Galler, and Morris Bernstein. Pattern matching for clone and concept detection. In *Reverse engineering*, pages 77–108. Springer, 1996.

[51] Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings of the 8th Working Conference on Reverse Engineering*, pages 301–309, 2001.

[52] Evgeny B. Krissinel and Kim Henrick. Common subgraph isomorphism detection by backtracking search. *Softw. Pract. Exper.*, 34(6):591–607, 2004.

[53] Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Polymorphic worm detection using structural information of executables. In *Recent Advances in Intrusion Detection*, pages 207–226. Springer, 2006.

[54] Harold W Kuhn. The hungarian method for the assignment problem. *Naval research logistics quarterly*, 2(1-2):83–97, 1955.

[55] Chao Liu, Chen Chen, Jiawei Han, and Philip S Yu. Gplag: detection of software plagiarism by program dependence graph analysis. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 872–881, 2006.

[56] Lannan Luo, Jiang Ming, Dinghao Wu, Peng Liu, and Sencun Zhu. Semantics-based obfuscation-resilient binary code similarity comparison with applications

to software plagiarism detection. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 389–400, 2014.

[57] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

[58] Andrian Marcus, Jonathan Maletic, et al. Identification of high-level concept clones in source code. In *Proceedings of the 16th International Conference on Automated Software Engineering*, pages 107–114, 2001.

[59] Jean Mayrand, Claude Leblanc, and Ettore M Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Proceedings of the 12th International Conference on Software Maintenance*, pages 244–253, 1996.

[60] Thomas J McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976.

[61] Brendan D McKay et al. *Practical graph isomorphism*. Department of Computer Science, Vanderbilt University Tennessee, US, 1981.

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

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

[64] Ginger Myles and Christian Collberg. K-gram based software birthmarks. In *Proceedings of the 20th ACM symposium on Applied computing*, pages 314–318, 2005.

[65] Beng Heng Ng and Aravind Prakash. Exposé: discovering potential binary code re-use. In *Proceedings of the 37th Computer Software and Applications Conference*, pages 492–501, 2013.

[66] Jeongwook Oh. Fight against 1-day exploits: Diffing binaries vs anti-diffing binaries. In *Blackhat technical Security Conference*, 2009.

[67] Jannik Pewny, Felix Schuster, Lukas Bernhard, Thorsten Holz, and Christian Rossow. Leveraging semantic signatures for bug search in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 406–415, 2014.

[68] Ashkan Rahimian, Raha Ziarati, Stere Preda, and Mourad Debbabi. On the reverse engineering of the citadel botnet. In *Foundations and Practice of Security*, pages 408–425. Springer, 2014.

[69] Anand Rajaraman, Jeffrey D Ullman, Jeffrey David Ullman, and Jeffrey David Ullman. *Mining of massive datasets*, volume 77. Cambridge University Press, 2012.

[70] Dhavleesh Rattan, Rajesh Bhatia, and Maninder Singh. Software clone detection: A systematic review. *Information and Software Technology*, 55(7):1165–1199, 2013.

[71] Chanchal K Roy, James R Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.

[72] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. Detecting code clones in binary executables. In *Proceedings of the 18th international symposium on Software testing and analysis*, pages 117–128, 2009.

[73] Madhu K Shankarapani, Subbu Ramamoorthy, Ram S Movva, and Srinivas Mukkamala. Malware detection using assembly and api call sequences. *Journal in computer virology*, 7(2):107–119, 2011.

[74] Joe Tekli, Richard Chbeir, and Kokou Yetongnon. Efficient xml structural similarity detection using sub-tree commonalities. In *Proceedings of the 22nd Brazilian symposium on Databases*, pages 116–130, 2007.

[75] Julian R Ullmann. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)*, 23(1):31–42, 1976.

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

[77] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449, 1981.

[78] Wuu Yang. Identifying syntactic differences between two programs. *Software: Practice and Experience*, 21(7):739–755, 1991.