ASSEMBLY CODE CLONE DETECTION FOR MALWARE BINARIES

Mohammad Reza Farhadi

A thesis
in
Concordia Institute for Information Systems Engineering

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

April 2013

Concordia University
School of Graduate Studies

This is to certify that the thesis prepared

By: **Mohammad Reza Farhadi**

Entitled: **Assembly Code Clone Detection for Malware Binaries**

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

Master of 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:

_____ Chair

Dr. Yong Zeng

_____ Examiner

Dr. Joey Paquet

_____ Examiner

Dr. Roch Glitho

_____ Supervisor

Dr. Mourad Debbabi

_____ Supervisor

Dr. Benjamin C.M. Fung

Approved _____
Chair of Department or Graduate Program Director

_____ 20 _____ _____

Dr. Robin A. L. Drew, Dean
Faculty of Engineering and Computer Science

Abstract

Assembly Code Clone Detection for Malware Binaries

Mohammad Reza Farhadi

Malware, such as a virus or trojan horse, refers to software designed specifically to gain unauthorized access to a computer system and perform malicious activities. To analyze a piece of malware, one may employ a reverse engineering approach to perform an in-depth analysis on the assembly code of a malware. Yet, the reverse engineering process is tedious and time consuming. One way to speed up the analysis process is to compare the disassembled malware with some previously analyzed malware, identify the similar functions in the assembly code, and transfer the comments from the previously analyzed software to the new malware. The challenge is how to efficiently identify the similar code fragments (i.e., clones) from a large repository of assembly code.

In this thesis, an assembly code clone detection system is presented. Its performance is evaluated in terms of accuracy, efficiency, scalability, and feasibility of finding clones on assembly code decompiled from real-life malware binary files and some DLL files from an Operating System. Experimental results suggest that the proposed clone detection algorithm is effective. This system can be used as the basis of future development of assembly code clone detection.

Acknowledgments

There are a number of people without whom this thesis might not have been written and fulfill the long road.

I would like to express my deepest gratitude to my advisors, Drs. Mourad Debbabi and Benjamin C. M. Fung, for their patience, enthusiasm, confidence on me to explore new research directions. I could not have imagined having better advisors for my Master's study.

My sincere thanks also go to my friends and staff at Concordia University and Computer Security Lab. I am grateful for the chance to be a member of the lab.

And last, but not least, I would like to express my heartfelt appreciation to my kind and warm family, who share my passions. My deepest appreciation goes to my parents for their persistent encouragement and unconditional love. I am also grateful to my siblings for their love and encouragements, specially my sister Leila, who is always nearby with supportive and inspiration thoughts.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Malware is any kind of software installed in computer systems without the owner's adequate consent that performs malicious activities. Malware includes, among others, computer viruses, worms, trojan horses, spyware and adware. Their malicious activities range from simple email spamming to sophisticated distributed denial of service attacks. New methodologies are used every day to create malware software that can be hidden from the lens of computer systems protective software.

Reverse engineering, although a time-consuming process, is often the primary step taken to gain an in-depth understanding of a piece of malware. To achieve a more efficient analysis, the analyst can manually compare the assembly code with a repository of previously analyzed assembly code and identify identical or similar code fragments. By identifying the matched code fragments and transferring the comments from the previous study to the new assembly code, the analyst can minimize her redundant effort and put more attention on the new part of the malware. Yet, the comparison process itself is also time-consuming, and successfully identifying similar code fragments often depends on the experience and knowledge of the analyst. In this thesis, an assembly code clone detection system based on the framework proposed by Sæbjørnsen et. al [50] is presented, and the performance of the system in terms of accuracy, efficiency, scalability, and feasibility of finding clones on assembly code decompiled from real-life binary and malware files is evaluated.

The problem of assembly code clone detection is informally described as follows: Given a large collection of previously analyzed assembly files and a specific target assembly file or a piece of target assembly code fragment, a user would like to identify

all code fragments in the previously analyzed assembly files that are syntactically or semantically similar to the target assembly file or the piece of target assembly code fragment. The challenges of the problem can be summarized as follows:

Simple keyword matching won't solve the problem: A simple method to identify the assembly code clone is to identify some keywords, such as constants, strings, and imports, in a code fragment, and then attempt to match them in other code fragments. Another alternative method is to perform a keyword search in RE-Google [1]. The keyword search capability is essential, yet insufficient, for assembly code clone detection because many code fragments do not contain any keywords or unique strings.

Large volume of data: The size of an assembly file can range from a couple of kilobytes to over dozens of megabytes of textual data. Depending on the user-specified parameters, each assembly file can be further decomposed into an array of regions with size proportional to the number of lines of an assembly file. The efficiency of a clone detection method refers to the period of time required for identifying all the clones. The scalability of a clone detection method refers to its capability of handling large collection of assembly code.

Syntactic and semantic clones: Two code fragments that are syntactically similar to each other are considered to be a syntactic clone. Two code fragments that perform the same computation but having different instructions are considered to be a semantic clone. Ideally, a clone detection algorithm should able to identify both types of clones. Yet, semantic clones are difficult to detect, especially in the context of assembly code.

The objectives of this thesis are (1) to introduce a formal framework for design and implementation of an assembly code clone detection, (2) to evaluate the feasibility of detecting exact clones with different levels of normalization, (3) to propose an accurate inexact clone detection, (4) to implement an efficient search capability on constants, strings, imports and code fragments, which are known to be valuable features for gaining insight into the binary files. Finally,

Contributions: The contributions of this thesis are summarized as follows. An assembly code clone detection framework based on Sæbjørnsen et al.'s work [50] is developed and significant extensions on its normalization and inexact clone detection method are made. The new normalization process improves the clone detection tool

by finding type I and type II clones. Sæbjørnsen et al. [50] presented an inexact clone detection method to identify clone pairs that are not exactly identical. The general approach is to first extract a set of features from each region and create a feature vector for each of them. Sæbjørnsen et al. used *locality-sensitive hashing (LSH)* to find the nearest neighbor vectors of a given query vector. Although this approach shows some encouraging results to identify clone pairs that were not exactly identical, it has an inappropriate assumption on the uniform distribution of vectors and the requirement of specifying precise probabilistic models. As the number of features increase (i.e., dimensions), the assumption on the uniform distribution of vectors may not hold. As a result, their approach may miss the vectors (clones) that are not uniformly distributed; therefore, increasing the chance of having false negatives. A new approach to address the issue of false negatives is presented and implemented in this thesis. This new approach can find Type III clones by an efficient feature extraction and filtering process algorithm. Finally, the framework is improved by storing the clone results in an XML file and visualizing the clones. Experimental results on real-life malware binaries obtained from the National Cyber-Forensics and Training Alliance (NCFTA) Canada suggest that our proposed clone detection algorithm is effective.

The thesis is organized as follows: Chapter 2 provides a literature review and background knowledge on source and assembly code clone detection. Chapter 3 formally defines the problem of assembly code clone detection. Chapter 4 discusses the implementation details and proposed algorithms used in the implemented assembly code clone detection tool. Chapter 5 presents the experimental results to illustrate the performance of the implemented assembly code clone detection system. Chapter 6 concludes the thesis and also suggests the potential extensions that can further improve the currently implemented system.

Chapter 2

Related Work and Background Knowledge

In this chapter, first, the state-of-art techniques for source code fragment matching are summarized to analyze the feasibility of applying the techniques to assembly code fragment matching. These techniques are categorized into eight groups. For each category, the most promising techniques are identified based on the analysis and experimental results obtained from different literature reviews, survey papers, and case study papers.

Then, the state-of-art techniques for assembly code fragment matching are summarized and categorized into six groups. The most promising techniques are also identified and briefly described.

The procedure of clone detection consists of two phases, transformation and comparison. The general idea is to transform the code into an intermediate format that facilitates more effective and efficient comparison.

Before describing the categorized techniques, the definition of code clone types with respect to textual and functional similarities is presented.

Clone Definition

Any two given similar code fragments can be similar in two different ways: textual and functional similarity. The following definitions of code clone types are widely used in the literature of clone detection and plagiarism (Roy et al. [48], [49]).

Textual Similarity:

- Type I: Identical code fragments except for variations in whitespace (perhaps

also variations in layout) and comments.

- Type II: Structurally/syntactically identical fragments except for variations in identifiers, literals, types, layout, and comments.
- Type III: Copied fragments with further modifications. Statements can be changed, added, or removed in addition to variations in identifiers, literals, types, layout, and comments.

Functional Similarity:

- Type IV: Two or more code fragments that perform the same computation that is implemented through different syntactic variants. This type of clone is also called a semantic clone.

2.1 Matching Source Code Fragment

Text-Based Approach

Text-based clone detectors consider the target source program as a sequence of lines or strings, and compare every pair of code fragments to find identical sequences of strings. Since the match is based purely on the text of the lexical approach, the identified clones often do not correspond to any structural elements of the language. Most text-based techniques do not transform the source code into some intermediate or normalized format, except for comments removal, whitespace removal, and some very basic preprocessing steps [23]. Text-based methods are robust to format alteration, but fragile to identifier renaming.

Johnson ([32], [33]) presented a pioneer work of a pure text-based approach that finds redundancy of code using fingerprints on a substring of the source code. The general idea of the method is to first calculate the signature of each line and then identify the matched substrings. First, a text-to-text transformation is performed on the source file for discarding uninteresting characters. Then, the file is divided into a set of substrings so that every character of the text appears in at least one substring. To compute the fingerprints of all length n substrings of a text, Johnson employed the *Karp-Rabin fingerprinting algorithm* [36] . A sliding-window technique in combination with an incremental hash function is used to identify sequences of lines having the same hash value as clones. For finding near-miss clones, his algorithm finds

a normalized/transformed text by removing all whitespace characters except for line separators and by replacing each maximal sequence of alphanumeric characters with a single letter, such as "a".

For example, a line

$$for(k = 1; k \leq n; k++)$$

is transformed to

$$a(a = a; a \leq a; a++)$$

and another line

$$\#define\ ABCD\ 123$$

is transformed to

$$\#aaa$$

Marcus and Maletic [46] searched for similarities of high-level concepts extracted from comments and source code elements. They employed *latent semantic indexing* [25] for finding similar code fragments in the source code. This information retrieval approach limits the comparison within comments and identifiers. Their results show that high-level, semantic clones can be detected with low computational cost. They also illustrated that their approach can be combined with other existing clone detection approaches. One major drawback is that this approach alone cannot detect two functions with similar structure and functionality if comments do not exist and the identifiers' names are different.

Ji et al. [31] introduced an adaptive local alignment method that considers the *frequencies of keywords* in the clone detection scoring function. The basic idea of adaptive local alignment is that the matching score of keywords should reflect the frequencies of keywords. Specifically, their method assigns a high score for a keyword with low frequency, and a low score for a keyword with a high frequency. The rationale is that it is rare to see keywords of low frequency being used by two fragments at the same time. Therefore, two programs using the same keywords of low frequency should be considered similar.

Token-Based Approach

A token-based clone detector parses and transforms source code to a sequence of tokens using compiler style lexical analysis. All the whitespaces, including line breaks, tabs, and comments between tokens are removed from the token sequence. A sequence is then scanned for duplicated subsequences of tokens, and finally, the original code

portions representing the duplicated subsequences are returned as clones. Compared with text-based approaches, a token-based approach is usually more robust against code changes such as formatting and spacing. Since variables of the same type are mapped into the same token, a token-based approach is usually more robust against identifier renaming. However, this approach is generally fragile to statement reordering and code insertion because it relies on sequential analysis. A reordered or inserted statement can break a token sequence that may otherwise be regarded as duplicate to another sequence. Token-based methods are also fragile to control replacement because, for example, *for* and *while* loops render different token sequences [44]

Kamiya et al. [34] proposed a multi-language token-based clone detection method called *CCFinder*, which extracts code clones in C, C++, Java, COBOL, and other source files. First, each line of source files is parsed into tokens by a lexer and the tokens of all source files are then concatenated into a single token sequence. The token sequence is then transformed based on the transformation rules of the language of interest, aiming at regularization of identifiers and identification of structures. For example,

```
void print_lines(const set⟨string⟩ &s) {
        int c = 0;
        set⟨string⟩::const_iterator i
                = s.begin();
        for (; i != s.end(); ++i) {
                cout << c << ", "
                        << *i << endl;
                ++c;
        }
}
```

is transformed to the following by some transformation rules.

```
void print_lines ( const set & s ) {
int c = 0;
const_iterator i
= s . begin ( );
for ( ; i != s . end ( ) ; ++ i ) {
cout << c << ", "
```

```
<< * i << endl;
++ c;
}
}
```

Then, all identifiers of types, variables, and constants are replaced with a special token, e.g., $\$p$. This identifier replacement makes code fragments with different variable names be clone pairs. For example, the above code will become the following after the replacements of identifiers:

```
$p $p ( $p $p & $p ) {
$p $p = $p ;
$p $p
= $p . $p ( ) ;
$p ( ; $p != $p . $p ( ) ; ++ $p ) {
$p << $p << $p
<< * $p << $p ;
++ $p ;
}
}
```

Burd and Bailey [18] conducted an independent experiment on CCFinder and found that CCFinder has a precision of 72% and recall of 72%, with a medium system of 16 KLOC. The recall rate of CCFinder is the best among all the tested methods in Burd and Bailey's experiments. Experimental results suggest that CCFinder is promising for Type I and Type II clone detections for source code.

Baker ([10], [9]) proposed another token-based method called *Dup*. As in CCFinder, Baker also used a lexer to tokenize the source code, then compared the tokens of each line using a suffix-tree based algorithm. Transformation rules on the token sequence were not applied in the work, and the notion of parametrized matching was introduced by a consistent renaming of the identifiers. Dup requires a lot of user interactions to determine whether tokens are identical. For example, the user may specify the same token ID to different data types, such as *int*, *short*, *long*, *float*, and *double*, depending on requirements.

The token-based techniques discussed above (Kamiya et al. [34]; Baker [9], [10]) make use of a *suffix* tree to detect similarities in the token string. Recently, researchers

have shown that an alternative data called *suffix* arrays (Manber and Myers [45]) can provide the same efficiency for string matching with much reduced space requirements (Abouelhoda et al. [6]). Another limitation of these token-based techniques is the separation of parameter and non-parameter tokens, which can potentially cause false negatives in clone detection.

To address the above shortcomings in a token-based approach, Basit et al. [12] proposed a method called *Repeated Tokens Finder (RTF)*. They made two contributions to the token-based approach: (1) RTF implements a flexible tokenization mechanism. (2) They consider the problem of clone detection as finding repeating substrings within the combined token string, and making the comparisons feasible by utilizing suffix arrays.

- *Flexible tokenization mechanism*: First, the language-specific tokenizer assigns a unique numeric ID to each token class of the source language, e.g., keywords, operators, comment markers, etc. Then, a single large token string is generated from all the source files. Identical segments of this token string are reported as clones that can be either exact clones (Type I) or parameterized clones (Type II) owing to the normalization of certain tokens according to the proposed tokenization. Next, RTF allows a user to suppress some insignificant token classes that may be considered noise in clone detection. For example, access modifiers, such as *private*, *protected*, and *public*, are not important for clone detection, and therefore should be suppressed. Furthermore, RTF allows a user to equate different token classes. For example, *int*, *short*, *long*, *float*, and *double* can be merged into the same ID, so that code fragments that differ only in data types become a clone pair.

- *Finding repeating substrings*: RTF treats clone detection as finding repeating substrings within a large combined token string. In fact, RTF finds the non-extendable repeating substrings because any non-empty subsequence of a clone is also a clone. Specifically, RTF applies (the method of Abouelhoda et al. [6]) to the problem of source code clone detection.

Most works on clone detection assume that the source database is static and the clone detection operation is performed in a batch mode. When the source database

is updated, the entire detection operation has to be performed again. Most literature in the field implicitly makes such an assumption, regardless of the program representation they operate on or the search algorithm they employ.

Recently, Hummel et al. [28] proposed a scalable token-based tool called *ConQAT* for detecting clones in the environment of an incrementally updated source database. ConQAT is open source. Its token-based clone detection method is more or less similar to the previously discussed methods, such as CCFinder and Dup, and ConQAT can detect Type I and Type II clones. The major contribution of ConQAT is its capability to efficiently identify the clones when updates are performed on the source database, by maintaining a data structure called *clone index*. It allows the lookup of all clones for a single file (and thus for the entire system), and can be updated efficiently when files are added, removed, or modified. For example, adding a new file may introduce new clones to any of the previously examined files and thus a comparison to all files is required if no additional data structure is used. The general idea of the clone index is similar to that of the inverted index used in document-retrieval systems in which a mapping is maintained between each word and all its occurrences. Similarly, the clone index maintains a mapping from sequences of normalized statements to their occurrences.

Tree-Based Approach

In the tree-based approach, the source code is first converted into an abstract syntax tree (AST) or a parse tree using a language-specific parser. Tree-matching techniques are then used to find similar subtrees, and the corresponding code segments are returned as clone pairs or clone classes. Variable names and literal values in the source may be abstracted in the tree representation, allowing for more flexible detection of clones. In general, tree-based approaches can detect Type I clones. For Type II clones, the token-based approach is more effective. Since this approach disregards the information about variables (in order to make code differing on variables' names appear the same on ASTs), it ignores data flow and is consequently fragile to statement reordering and to control replacement (Liu et al. [44]).

Baxter et al. [13] presented an AST-based clone detection method called *CloneDr*. First, a language-specific parser generates an annotated parse tree. See Figure 1 for an example

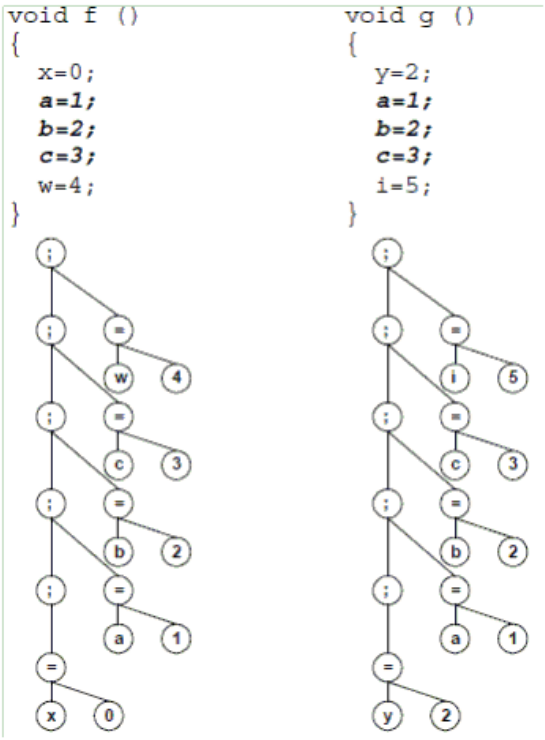


Figure 1: Annotated Parse Tree (Baxter et al. [13])

The next step is to find the subtree clones by comparing every subtree to other subtrees for equality. Subtrees are compared using characterization metrics based on a hash function and tree matching. This approach works for detecting Type I clones. When locating Type II clones, hashing on complete subtrees fails precisely because a good hashing function includes all elements of the tree and, therefore, considers trees with minor differences to be mismatched. Baxter et al. [13] solved the problem by choosing an artificially "bad" hash function. This function must be characterized in such a way that the main properties one wants to find on Type II clones are preserved.

Type II clones are usually created by copying and pasting code fragments, followed by making small modifications. These modifications usually generate small changes to the shape of the tree associated with the copied piece of code. Therefore, they argue that this kind of Type II clone often has only some different small subtrees. Based on this observation, a hash function that ignores small subtrees is a good choice. They used a hash function in their experiments that ignores only the identifier names (leaves in the tree). Thus, their hashing function puts trees that have similar modulo identifiers into the same hash bins for comparison. They further proposed some threshold-based similarity measures to measure the similarity of two subtrees.

Wahler et al. [58] proposed a method to find Type I and Type II clones at a more abstract level than an AST. A program AST is converted to an XML representation and a frequent itemset technique is applied to the XML representation of to find clones. They modified the frequent itemset technique to fit it to the problem of code clone detection.

Evans and Fraser [26] proposed a further abstraction, known as structural abstraction, of a program's AST for finding both Type I and Type II clones with gaps. While ASTs are built from a lexical abstraction of a program by parameterizing only AST leaves (abstracting identifiers and literal values), structural abstraction is obtained by further parameterizing the arbitrary subtrees of ASTs. For example, $x = a[?]$ lexically matches $x = a[i]$ because it includes only a leaf and a unary node that identifies the type of the leaf. $x = a[?]$ also structurally matches $x = a[i+1]$ because it includes a binary AST node.

Similarity Distance-Based Approach

Brixtel et al. [16] presented a comprehensive similarity distance-based plagiarism detection framework. In this section, the only elaborated steps are the ones that are relevant to the problem of code fragments matching, namely normalization, segmentation, and similarity measure. Brixtel et al. do not present a predefined method for segmentation. A code segment can be a line or a block of code. The distance of two segments, s_1 and s_2, is measured by a distance function $Dist(s_1, s_2)$.

The distance function can be the *edit distance*, e.g., Hamming or Levenshtein (Levenshtein [43]), by counting the number of operations (insertions, deletions, and replacements) required to transform s_1 to s_2. Another possible distance function is the *information distance*, which can be approximated using data compression, e.g., gzip. Let $c(s)$ be the compressed version of a segment, and let $|c(s)|$ be the size of the compressed version. The distance between two segments (or two strings) is:

$Dist(s_1, s_2) = 1 - \frac{|c(s_1)|+|c(s_2)|-|c(s_1,s_2)|}{max(|c(s_1)|,|c(s_2)|)}$

where $|c(s_1, s_2)|$ denotes the compressed size of the concatenation of s_1 and s_2. Refer to (Cilibrasi and Vitanyi [20]) for the details of this formula.

Program Dependency Graph (PDG)-Based Approach

Program Dependency Graph (PDG)-based approaches attempt to identify a higher abstraction of a source code than the previously discussed approaches by considering the semantic information of the source code. PDG-based approaches use nodes to

represent expressions and statements, while the edges represent control and data dependencies. This representation abstracts from the lexical order in which expressions and statements occur to the extent that they are semantically independent. Intuitively, PDGs encode the program logic and in turn reflect developers' thinking when code is written. The search for clones is then turned into the problem of finding isomorphic subgraphs (Roy and Cordy [48]).

Krinke [41] proposed a clone detection method based on fine-grained program dependence graphs. His method, commonly known as *Duplix* (Roy and Cordy [48]), considers both the syntactic structure and data flow of a software program. The first step is to transform code fragments into fine-grained PDGs, and the second step is to identify similar subgraph structures from the PDGs. The vertices of the fine-gained PDG represent components of expressions. There are three types of edges between the vertices. *Immediate control dependence edges* are between the components of an expression, which are evaluated before the source code is evaluated. *Value dependence edges* represent the data flow between the expression components. *Reference dependence edges* represent the assignment of values to variables.

Komondoor and Horwitz [39] proposed a PDG-based clone detection method called *PDG-DUP*. The general idea is to first partition all PDG nodes based on the syntactic structure of the statement/predicate that the node represents. For each pair of matching nodes (r_1, r_2), PDG-DUP identifies the isomorphic subgraphs of the PDG that contains r_1 and r_2 using backward and forward slicing. The identified isomorphic subgraphs are clones.

Liu et al. [44] developed an *efficient* PDG-based plagiarism detection algorithm called *GPLAG*. They employed a relaxed subgraph isomorphism comparison by introducing a relaxation threshold γ . Two subgraphs g and g' are γ-isomorphic if their difference is within the threshold γ. Although subgraph isomorphism comparison is in general NP-complete, they developed GPLAG based on three observations, making GPLAG a feasible real-life application. First, PDGs cannot be arbitrarily large as procedures are designed to be of reasonable size for developers to manage. Second, PDGs are not general graphs and their peculiarity, like varieties of vertex types, makes backtrack-based isomorphism algorithm efficient. Finally, unlike traditional isomorphism comparison, a user usually is satisfied as long as one, rather than all, isomorphism between g and g' is found. A user can look into the details once a pair

of matched procedures has been identified. These three factors make isomorphism comparison efficient in GPLAG.

Liu et al. identified five categories of source code alterations and considered their effect on the PDG.

1. Format alteration
2. Identifier renaming
3. Statement reordering
4. Control replacement
5. Code insertion

The first and second alteration remove blanks, insert comments, or rename the identifiers in the source code. Hence, these alterations do not alter the PDG because they do not affect the dependencies of the graph. Statement reordering changes the location of statements without causing error. This type of alteration also does not alter the PDG because the statements can be reordered if they do not have dependencies. Control replacement changes the control flow of the source code. For example, a *for* loop can be equivalently replaced by a *while* loop. This alteration can add a new program vertex to the subgraphs but does not affect the dependencies of subgraphs. The last alteration is code insertion, which inserts immaterial code into the program so that the original program logic does not change. These types of alterations can introduce new dependencies into the program but they are not supposed to interfere with existing dependencies.

Let n be the number of procedures in the first source code file and m be the number of procedures in the second source code file. To identify the clone pairs, Liu et al. observed that it is not necessary to perform a full isomorphism comparison for $n * m$ pairs for PDGs. Most PDG pairs can in fact be excluded from detailed isomorphism testing because they are dissimilar, even with a high-level examination. Therefore, they proposed a *lossy filter* to prune these dissimilar PDG pairs. Unlike conventional similarity measurement, usually based on a certain distance metric, this filter follows a similar reasoning to hypothesis testing: A PDG pair (g, g') is preserved until enough evidence is collected against the similarity between g and g'. In comparison

with distance-based methods, this approach avoids the difficulty of proper parameter setting and also provides a statistical estimation of the false negative rate. Their experiment shows that GPLAG can efficiently prune 90% of the original search space.

Metrics-Based Approach

Most of the previously discussed approaches identify clones by directly comparing code. In contrast, metrics-based approaches gather different metrics for code fragments and compare the metrics vectors. There are several clone detection techniques that use various software metrics for detecting similar code. First, a set of software metrics called *fingerprinting functions* are calculated for a class, a function, a method or even a statement, and then the metrics values are compared to find clones over these syntactic units. In most cases, the source code is parsed to its AST/PDG representation for calculating such metrics.

Kontogiannis et al. [40] proposed two approaches to detect clones. The first approach is named *direct comparison of metrics values* that classifies a code fragment in the granularity of *begin-end* blocks. Two code fragments are considered similar if their corresponding metrics values are close. The second approach uses a *dynamic programming* technique for comparing *begin-end* blocks at the statement level.

Program features relevant for clone detection focus on data and control flow properties. Kontogiannis et al. used the following metrics:

1. The number of functions called (fanout).
2. The ratio of input/output variables to the fanout.
3. McCabe cyclomatic complexity.
4. Modified Albrecht's function point metric. And,
5. Modified Henry-Kafura's information flow quality metric.

Let s be a code fragment.

$$McCabe(s) = e - n + 2,$$

where e is the number of edges in the control flow graph, and n is the number of nodes in the graph.

$$Albrecht(s) = p_1 * VARS\_USED\_AND\_SET(s)+$$
$$p_2 * GLOBAL\_VARS\_SET(s)+$$
$$p_3 * USER\_INPUT(s)+$$
$$p_4 * FILE\_INPUT(s)$$

where p_1, p_2, p_3 and p_4 are weight factors.

$$Karfura(s) = (Kafura\_in(s) * Kafura\_out(s))^2,$$

where *Karfura_in(s)* is the sum of local and global incoming dataflow to the code fragment s, and *Karfura_out(s)* is the sum of local and global outgoing dataflow to the code fragment s.

Mayrand et al. [47] proposed a method called *CLAN* that calculates 21 functional metrics, e.g., number of lines of source, number of function calls contained, and number of CFG edges, for each function unit of a program. Units with the similar metrics values are identified as code clones. Partly-similar units are not detected. It uses a representation of the source code called *Intermediate Representation Language (IRL)* to characterize each function in the source code. For example, Figure 2 depicts the IRL representation of the following function:

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

Metrics are calculated from names, layouts, expressions, and control flows of functions. A clone is defined only as a pair of whole function bodies that have similar metric values. This approach does not detect clones at other granularity levels such as segment-based clones, which occur more frequently than function-based clones. Very similar kinds of method-level metrics, such as the number of calls from a method, number of statements, McCabe's cyclomatic complexity, number of use-definition of

non-local variables, and number of local variables, are features for identifying similar methods.

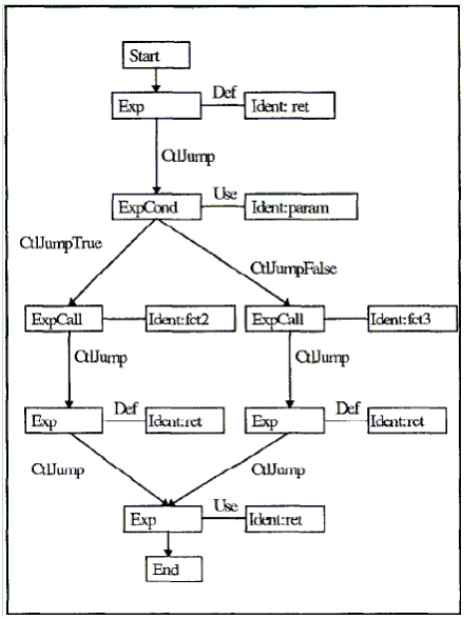


Figure 2: IRL Representation of the Sample code (Mayrand et al. [47])

Shawky and Ali [55] performed experiments on other metrics values: (1) the number of inputs a function uses (2) the number of outputs (3) the number of declarative and executable statements (4) the average number of lines containing source for all nested functions (5) the number of edges, nodes, and connected components, and (6) the maximum nesting level of control constructs in a function.

Memory-Based Approach

Memory-based clone detection is a recently proposed approach by Kim et al. [38]. Unlike most of the previously described approaches that are based on textual or structural similarity, the proposed *Memory Comparison-based Clone Dectector (MeCC)* identifies clones by comparing programs' abstract memory states, which are computed by a semantic-based static analyzer. The primary objective is to detect gapped clones (Type III) and semantic clones (Type IV). The clone detection ability of MeCC is independent of the syntactic similarity of clone candidates. MeCC has two phases: (1)its uses a path-sensitive semantic-based static analyzer to estimate the memory states at each procedure's exit point, and (2) compares the memory states to identify clones. Figure 3 shows an overview of MeCC.

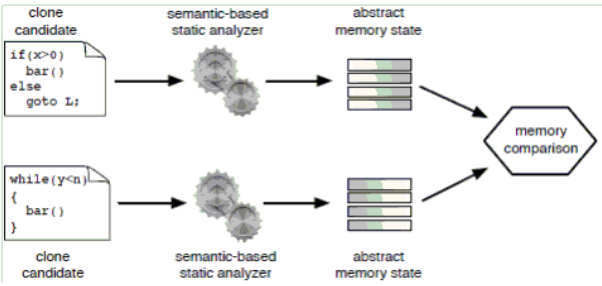


Figure 3: Overview of MeCC (Kim et al., [38])

```
1  int* foo(list *a,
2           int b){
3    int res = 0;
4    if (a->len > 5)
5       res = bar(b);
6    return res;
7  }
8  int* bar(int x){
9    int *m = 0;
10   if (x > 0)
11      m = malloc(x);
12   return m;
13 }
```

The abstract memory state at line 6

a	$\{\langle \text{true}, \alpha \rangle\}$
α.len	$\{\langle \text{true}, \beta \rangle\}$
b	$\{\langle \text{true}, \gamma \rangle\}$
res	$\{\langle \beta > 5 \wedge \gamma > 0, \ell \rangle,$ $\langle \beta \leq 5 \vee (\beta > 5 \wedge \gamma \leq 0), 0 \rangle$ $\}$

The procedural summary of bar

$x > 0$ return alloc

$x \leq 0$ return 0

Figure 4: Example of Abstract Memory State (Kim et al. [38])

Figure 4 shows the abstract memory states of the sample programs *foo* and *bar*. The joined memory state at the return point of *foo* (line 6) is shown as the table in Figure 4.

```
1 int* foo2(list2 *x,
2           int y){
3   int ret = 0;
4   if (x->val > 5 && y > 0)
5     ret = malloc(y);
6   return ret;
7 }
```

The abstract memory state at line 6

x	$\{\langle \text{true}, \alpha \rangle\}$
α.val	$\{\langle \text{true}, \beta \rangle\}$
y	$\{\langle \text{true}, \gamma \rangle\}$
ret	$\{\langle \beta > 5 \wedge \gamma > 0, \ell \rangle,$ $\langle \beta \leq 5 \vee \gamma \leq 0, 0 \rangle$ $\}$

Figure 5: Example of Abstract Memory State Comparison (Kim et al. [38])

Figure 5 shows another procedure called *foo2*, which is a semantic clone of procedure *foo* in Figure 4. Ignoring the names of variables, symbols, field variables, and variable types, MeCC can determine that $\beta \leq 5 \vee \gamma \leq 0$ and $\beta \leq 5 \vee (\beta > 5 \wedge \gamma \leq 0)$ are in fact equivalent and therefore, the two fragments are clones.

Hybrid Approach

Different approaches have different strengths and weaknesses. The idea of the hybrid approach is to use different approaches with the goal of combining their strengths

and avoiding their weaknesses. In this section, some papers that combine token-based or text-based approaches with tree-based or PDG-based approaches are reviewed to get the beneficial aspects of multiple approaches. This work can lead to analyzing large scale source codes and software with better precision. Also, it can increase the portability of syntactic and semantic approaches to assembly code clone detection.

Balazinska et al. [11] proposed a hybrid approach of characterization metrics and DPM (Dynamic Pattern Matching). Their paper discusses only the detection of whole methods, although the approach may also be applied to detect partial code fragments. Characteristic metrics valued are computed for each of the method bodies and compared to find clusters of similar methods. The token sequences for each pair of similar methods are then compared by the Dynamic Pattern Matching Algorithm of Kontogiannis et al., in order to identify cloned methods. Finally, the found cloned methods are classified into 18 categories.

Tairas et al. [56] developed a plug-in for the Microsoft Phoenix framework for automatic clone detection. They used AST nodes to generate a suffix tree, allowing analysis on the nodes. A path is made from the root to a leaf for each suffix of a string. This is done by evaluating each character in the suffix and generating new edges when there are no existing edges that represent the character in the suffix tree. A suffix tree is useful in string matching because duplicate patterns in the suffixes will be represented by a single edge in the tree.

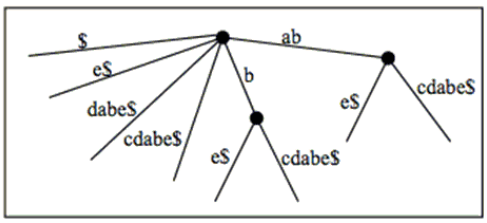


Figure 6: "abcdabe$"'s Suffix Tree (Tairas et al. [56])

Figure 6 depicts the suffix tree for the string "abcdabe$". The non-empty suffixes sorted by alphabetical order are:

abe$

abcdabe$

bcdabe$

be$

vcdabe$

dabe$

e$

$

The pattern *ab* is represented by a single edge. Two suffixes *abcdabe$* and *abe$* pass through this edge. The split at the end of this edge continues the two suffixes where the next character differs between them. By looking at the suffixes passing through the edge that represents the pattern *ab*, the location of this string pattern can be determined. Using the method for searching for certain edges in the suffix tree, duplicated functions can be determined.

Zeidman ([61], [62]) published two similar patents for detecting plagiarism in source code at the file level. His invention fully compares the features of each pair of source code. Each source file is represented by three arrays: an array of source lines, an array of comment lines, and an array of identifiers. His methodology employs five measures to identify clones: source line matching, comment line matching, word matching, partial word matching, and semantic sequence matching. Figure 7 illustrates the array of source lines, the array of comment lines, and the array of unique identifiers of the function named *fdiv*. Zeidman provided an equation for each of the five measures and the similarity of two source files is the weighted sum of the five measures.

```
/*   ---- begin    routine ----   */
void fdiv(
    char    *fname,      // file name
    char *path)          // path
{
    int Index1, j;

    while (1)
        j =    strlen(fname);
    /* find the file     extension */
```

(a) C source code snippet

```
SourceLines1[0] = ""
SourceLines1[1] = "void fdiv"
SourceLines1[2] = "char fname"
SourceLines1[3] = "char path"
SourceLines1[4] = ""
SourceLines1[5] = "int Index1 j"
SourceLines1[6] = ""
SourceLines1[7] = "while 1"
SourceLines1[8] = "j strlen fname"
SourceLines1[9] = ""
```

```
CommentLines1[0] = "---- begin routine ----"
CommentLines1[1] = ""
CommentLines1[2] = "file name"
CommentLines1[3] = "path"
CommentLines1[4] = ""
CommentLines1[5] = ""
CommentLines1[6] = ""
CommentLines1[7] = ""
CommentLines1[8] = ""
CommentLines1[9] = "find the file extension"
```

(b) Source code and comment line arrays

```
Word1[0] = "fdiv"
Word1[1] = "fname"
Word1[2] = "path"
Word1[3] = "Index1"
```

(c) Array of unique identifiers (non-keywords)

Figure 7: Arrays of Source Lines, Comment Lines, and Unique Identifiers (Zeidman [61])

Davis and Godfrey [22] proposed a novel technique to detect clones by analyzing the assembly-version of source code. Their work complements token- and text-based techniques for source code clone detection. First, the source code is normalized and is reduced to a sequence of simple operations. Then, it is compiled into assembly code. The comparison is performed on assembly instructions in functions. A search-based approach is used to find maximal pairings of distinct matched assembly code instructions from two distinct assembly subsequences that occur in the same sequence. The search-based algorithm examines each instruction by walking through each function and each array of instructions within a function. Given two sequences of comparable instructions P and Q, the comparison process compares every pair of instructions in P and Q, assigns a positive weight to each matched pair, and assigns a negative weight for each unmatched pair. Consequently, the subsequences with high sum of weights are considered to be clones.

Keivanloo et al. [37] proposed a hybrid approach to improve the scalability of the current existing clone detection tools. They devise a *divide-and-conquer* algorithm

that splits a given data set into n random subsets and extracts clones from each subset independently. The drawback is that the algorithm will miss the clone pairs that span across two different subsets. Their algorithm provides a trade-off between recall and scalability.

2.2 Matching Assembly Code Fragments

Text-Based Approach

A text-based approach in binary clone detection considers the executable portion of a binary as a sequence of bytes or lines of assembly code and compares every pair of code segments to find identical sequences.

Jang et al. ([29], [30]) proposed a fingerprinting algorithm called *BitShred* based on bloom filters to cluster malware samples. In addition, they stated that BitShred can be used to detect software bugs resulting from copied code. However, they did not have any success in their experiments.

BitShred consists of three phases: shredding a file, creating a fingerprint, and comparing fingerprints. In the shredding phase, BitShred divides all executable code sections into fragments called *shreds* (i.e., n-grams). Figure 8 shows an example of shredding a byte sequence with n= 5.

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 8: Shredding a Byte Sequence with n = 5 (Jang and Brumley, [29])

Next, to improve storage efficiency and scalability, BitShred uses a *Bloom filter* (Bloom, [14]) created from all shreds of a given file to represent the fingerprint of the file. Then, BitShred calculates similarity between two fingerprints by using the *Jaccard index*, which is defined as the size of the intersection divided by the size of the union of the sample sets. In the case of comparing two Bloom filters, this is reduced to the ratio

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

where *S(BF)* is the total bits set in the Bloom filter *BF*.

Finally, files having a similarity score higher than a threshold t can be clustered together. Clustering is performed in two steps: first, malware samples with similarity equal to 1 are clustered. Similarity between every pair of the remaining samples is then calculated.

Token-Based Approach

Token-based binary clone detectors disassemble executable code segments and partition lines of assembly into *opcodes* and *operands*. Opcode and operand types may be generalized or filtered and the resulting sequence is scanned to find duplicates.

Schulman ([52], [53], [54]) whitelists "boilerplate" code in binaries to be analyzed for litigation purposes. To address this, he designed a system to create a database of previously analyzed binaries to recognize duplicate functions. This appears to be the first work on detecting code clones at the functional level. In addition, this work was designed to handle a large volume of assembly code.

The basic idea of the system is to create a hash of each function in the binary being analyzed and to store it in the database. Identical hash values that occur in more than one binary indicate a code clone. This approach will detect some functions whose source code is identical, but will miss many others. Compiling source code does not always produce an identical assembly each time. Stack memory addresses may change, depending on the functions position in the binary. In addition, the binary may be compiled as Unicode versus ASCII, resulting in different assembly instructions. The source code may also have modifications such as different hard coded values in its variables, even though prior to this modifications the source was identical. To address these changes in the resulting assembly, the author normalizes the instructions in the following manner:

1. Only the opcode, not the operands, are used in the hash.

2. Opcodes are converted to their mnemonics. For example opcodes 51 and 52 are instructions for *push ecx* and *push edx*. The hash is now performed on the resulting generic opstring *push* instead of their numeric counterparts.

3. Location labels are added to the opstring as *loc* to add structure information for a more accurate hash.

4. Windows API calls are included in a normalized format (again, wide vs. ASCII formats are ignored).

5. Commonly occurring mnemonics such as *mov*, *pop* and *push* are ignored to further handle minor code changes.

An example of the resulting opstring for a function is:

loc,[MessageBox],cmp,jnz,ret

This string is hashed and stored in the database.

Karim et al. [35] addressed the problem of classifying new malware into existing malware families whose individual entries share common code. Their goal has been to create phylogeny models of malware based on features of their binary code, such as biologists create phylogeny models based on nucleotide, protein, and gene sequences of organisms. They built the models to handle program evolution through code rearrangements (instruction or block reordering). They experimented with an *n-gram* comparison approach as well as its permutations, which they refer to as *n-perms*. They argueed that permutations could include instruction, block, or subroutine reordering. In such situations, the reordering can make sequence-sensitive techniques, such as n-gram, produce undesirable results if they report similarity scores that are too low for reordered variants or descendants. Experiments were performed on artificially constructed permutations of worms as well as unrelated samples. This work evaluates the use of n-grams versus n-perms for clustering malware samples into their respective families.

The first step is to use the tokenizer to transform an input program into a sequence of opcodes. Next, n-grams and n-perms are extracted from this sequence. An n-perm would simply be an ordered n-gram. Subsequently, they create a feature occurrence matrix with entries *i, j* that refer to the number of times a feature (i.e., a specific n-gram or n-perm) occurs in program *j*. Then, they create a symmetric similarity matrix where each *i, j* entry is the calculated similarity between programs i and j. Similarity is calculated by using the $tf * idf$ weighting (van Rijsbergen, [57]) and *cosine similarity measure* (Zobel and Moffat, [63]). This method makes features that are common among many programs weighted lower and those features common within a program weighted higher.

tokenizer $\rightarrow$ *feature occurrence matrix extractor* $\rightarrow$ *similarity metric calculator*

Walenstein et al. [59] presented a mechanism called *Vilo* to search a database of previously analyzed malware binaries using a new malware sample as the search string. They adapted techniques from their previous work (Karim et al., [35]) to perform the similarity matching for the search.

In the Vilo method, whole programs are compared using "n-grams" and "n-perms" extracted from disassembled binaries that have been unpacked and unencrypted. They used a vector model for comparison. These extracted features are counted, weighted, and converted into vectors that are then compared using their cosine similarity measure (Zobel and Moffat, [63]). To perform a search using the Vilo method, a new sample is compared to each existing feature vector in the database.

Metrics-Based Approach

Sæbjørnsen et al. [50] presented a general *clone detection framework* that operates on binaries. It utilizes an existing tree similarity framework, models the assembly instruction sequences as vectors, and groups similar vectors together using existing "nearest neighbor" algorithms.

They first disassemble the input binaries by using a disassembler, such as IDA Pro [2], and then create intermediate representations of the assembly code. Then, a binary is partitioned into overlapping code segments that consist of a block of contiguous assembly extracted from within a function. These regions are created using a sliding window. Their method then creates a normalized instruction sequence, abstracting the information of memory location and registers. Next, it performs clone detection on the normalized structure sequence. They define two methods for creating clone clusters. The first method is an *exact match* that uses a hash for each code region, and a clone exists if there are any repeated hash values. The second method is an *inexact match*, which extracts a set of features from a code region and looks for other code regions with the same feature set. They count the number of occurrences of each feature to create a feature vector for each region. Next, they use *locality-sensitive hashing (LSH)* (Andoni and Indyk, [7]) on each region and perform a distance calculation for clustering based on features for inexact matching.

Sæbjørnsen et al. utilized IDA Pro for disassembling the code. However, the rest of the implementation has no reliance on it. After the disassembly, the instructions are put into the *ROSE intermediate representation* (Schordan and Quinlan, [51]). The normalized code regions and feature vectors are then extracted and stored in a

SQLite database. Finally, both the exact or inexact match detectors run on top of the database.

Bruschi et al. [17] presented a technique to normalize assembly code to detect polymorphic and metamorphic malware. The main objective of this work is based on using a normalization technique to ease the comparison between malware samples. The authors developed a prototype implementation on top of *Boomerang*, an open source decompiler. The normalization is the process of transforming a piece of code into a canonical form. Bruschi et al. used the following normalization techniques:

- Instruction meta-representation: This is a high-level representation of machine instructions that mimics the semantics of assembly language (opcode, registers, memory address, and flags).

- Propagation: It is used to propagate forward values assigned or computed by intermediate instruction. Once an instruction defines a value, this variable is used by subsequent instructions without being redefined. The purpose is to generate high level expressions and to eliminate all temporary variables. The following list illustrates a propagation of values in high-level representation of assembly code:

Before Propagation	**After Propagation**
r10 = [r11]	r10 = [r11]
r10 = r10 | r12	r10 = [r11] | r12
[r11] = [r11] & r12	
[r11] = ~[r11]	
[r11] = [r11] & r10	[r11] = (~([r11] & r12)) & ([r11] | r12)

- Dead code elimination: This technique consists of removing instructions that are never used.

- Algebraic simplification: It simplifies arithmetical and logical operations.

- Control flow graph compression: A control flow can be heavily modified by inserting fake conditional and unconditional jumps. As a result, its compression is necessary to ease browsing and analysis.

Bruschi et al. stated that it is not possible to compare samples by matching byte by byte in their formalized form. They adopted a method to measure the similarity

Metrics
m1: number of nodes in the control flow graph.
m2: number of edges in the control flow graph.
m3: number of direct calls.
m4: number of indirect calls.
m5: number of direct jumps.
m6: number of indirect jumps.
m7: number of conditional jumps.

Table 1: Metrics (Bruschi et al [17])

between different pieces of code. The method was introduced by (Kontogiannis et al. [40]). A code is characterized by a vector of metrics' values and a measure of distance between different code fragments. Table 1 shows the metrics employed by Bruschi et al.

These metrics identify the fingerprint of a code fragment as a 7-tuple (m1, m2, m3, m4, m5, m6, m7). The fingerprint is used for code fragments comparison. For the code fragments "a" and "b", the comparison is represented by the Euclidean distance: $\sqrt{\sum(m_{i,a} - m_{i,b})^2}$ for $i = 0 \ldots 7$, where $m_{i,a}$ and $m_{i,b}$ are the i^{th} metrics calculated on code fragments "a" and "b".

Structural-Based Approach

In software engineering, a program is considered as a set of components or code blocks that are built to achieve computational tasks within computers and calculators. A program consists of a set of instructions written in high- or low-level languages and structured within blocks. The structural analysis of software is a process that is used to characterize the execution schema of a program. Furthermore, it allows retrieval of code information by browsing different parts of the code. The structural analysis can be used to detect bugs within programs and to make comparisons between programs. In this thesis, the second issue is of primary interest. Different structural analysis techniques for code similarity detection will be discussed in this section, as well as some works that are related to the detection based on structural analysis approach.

Dullien et al. [24] presented the results of research on executable code comparison for attacker correlation. They implemented a system that can identify code similarities in executable files. Dullien et al. based their approach on the structural comparison of executable files. The goal of the system is twofold: querying for a particular feature within malicious code and recognizing similarities between

different malicious pieces of code. In order to query a large set of data, the authors implemented a function that hashes different control flow graphs. Specifically, they converted the set of edges in a control flow graph into a set of n-tuples of integers. A suitable encoding of each set is then constructed as follows: Let M be the set of all control flow graphs and E_g be the set of edges of a particular G belonging to M. The graph is represented with the following function:

$$tup: \mathrm{M} \rightarrow \mathfrak{P}(\mathbf{Z}^5)$$

$$tup(g) \rightarrow \left\{ \begin{pmatrix} \square \\ \square \\ topologicalorder(src(e)) \\ indegree(src(e)) \\ outdegree(src(e)) \\ indegree(dest(e)) \\ outdegree(dest(e)) \end{pmatrix} \mid e \in E_g \right\}$$

This function outputs a set of integers that represents a given graph in a simple manner. Next, in order to encode the set of integers, the authors chose a Q_5 vector $\left[1, \sqrt{2}, \sqrt{3}, \sqrt{5}, \sqrt{7}\right]$ to convert integer tuples into a real number:

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

The hash, called *MD-index*, is computed by the following function:

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

where $t \in tup(g)$

Dullien et al. developed a matching algorithm based on the approximation to maximum sub-graph isomorphism problem. The algorithm attempts to map between nodes and edges by observing a list of characteristics:

- Byte hash: a traditional hash over the bytes of the function or a basic block
- MD-index of a particular function
- MD-index of source and destination of call-graph edges
- MD-index of graph neighborhood of a node/edge (a sub-graph that is originated from a given node or edge)
- Small prime product: a simple way to compute a hash of mnemonics sequence. This method ignores compiler-induced reordering of instructions.

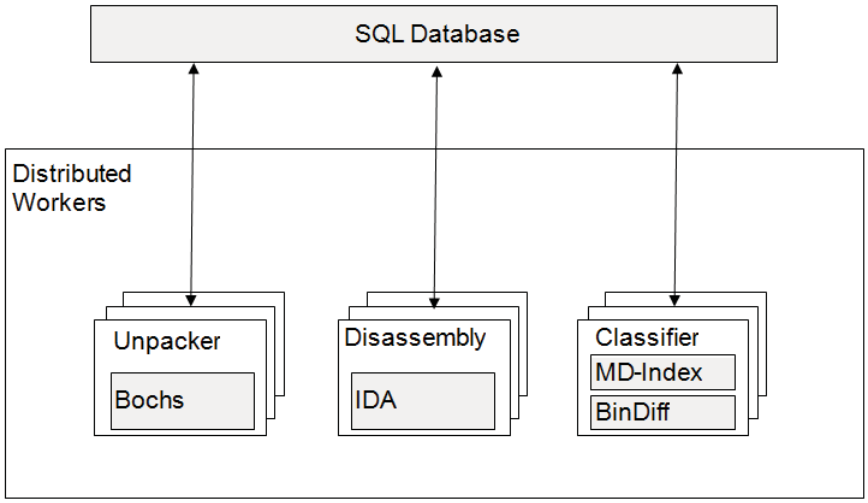


Figure 9: System Architecture (Dullien et al. [24])

Figure 9 shows the architecture of the system, where different components fetch data to perform desired computations. The system consists of four components:

- *Unpacker* is an unpacking component that removes encryption and obfuscation. The system monitors the statistical properties of the memory. Once the memory pages drops, the components assume that encryption/compression was removed. It writes memory dumps into the database.
- *Disassembly* extracts the control flow graphs and call-graphs.
- *Scheduler* performs a rough comparison based on MD-indices of functions in disassembly. It reduces the number of functions and blocks, which are susceptible to be compared with analyzed chunks of a given binary.
- Comparison component performs the comparison algorithm by querying the database. It writes the result of comparison back to the database.

Carrera and Erdlyi [19] addressed the challenge of having a large number of malware samples. They developed a system based on graph theory to rapidly and automatically analyze and classify malware based on its underlying code structure. This work is the predecessor to (Briones and Gomez, [15])'s and shares much in common with its underlying theory and implementation. After IDA Pro is used to disassemble a binary, their tool exports a subset of the information into the *Reverse Engineering Markup Language (REML)*. Carrera has continued with his work

on this tool and produced *http://dkbza.org/idb2reml.html*. Using this tool, they use *http://dkbza.org/pyreml.html* to create a pythonic representation of the data that can be queried during the comparison.

They used operating system and library calls in creating their adjacency matrices. Here their work differs from (Briones and Gomez, [15]) who also used functions that had similar CFG features. However, after their algorithm exhausts matches, they do apply CFG signatures similar to the other paper.

Briones and Gomez, [15] designed an automated classification system for binaries with a similar internal structure. They used graph theory to identify similar functions that are used to classify malware samples. Their objective is to classify new samples to previously analyzed malware families, thus reducing reverse engineering efforts. Although their objective is different from the problem of matching code fragment studied in this thesis, the way they identify similar functions is relevant.

The input for their comparison algorithm is an unpacked binary disassembled with IDA Pro. The next step is to model binaries into adjacency matrices. For example, in Figure 10, each directed edge represents one function calling another and can be modeled as the following adjacency matrix. The rows represent the calling functions and the columns the called functions. To build these adjacency matrices, the algorithm identifies fixed points that consist of known API calls, function hashes, and function call graphs (CFGs).

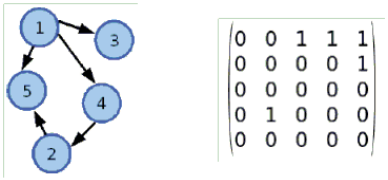


Figure 10: Directed Graph and Adjacency Matrix (Briones and Gomez, [15])

To compare two binaries, their method creates an adjacency matrix for each binary using identified fixed points as the columns and unidentified functions as the rows. These initial adjacency matrices have identical columns. They determine known API calls as those identified by IDA Pro that are shared between the binaries. The function hashes are unique CRC32 hashes shared between the binaries. Finally, the functions CFGs are those that have a unique tuple with a minimum Euclidean distance. A tuple for a CFG consists of a number of basic blocks, a number of edges, and a number of subcalls.

The comparison algorithm, from each matrix, searches for the functions that have identical column entries and are unique in their respective matrices. These identical functions are removed from their matrices and are added as a column to both matrices. This process continues until no more matches are found. The remaining rows are unidentified functions with no match or functions that do not have a unique match. Their method can potentially match more functions by adding further fixed points that initially were not unique and by starting the match process over again. They defined similarity as a ratio of matched functions to total functions, where 0 indicates no matches and 1 indicates a complete match.

Implementation: Graph comparison on binaries is a computationally intensive process. In their implementation, they suggested some techniques to improve efficiency. First, they did not rely on IDA Pro for disassembly for efficiency reasons. Second, when comparing rows they first split the row into blocks of 32 bits for comparison.

They applied filtering when analyzing new samples to reduce the number of comparisons. They filtered on file size, compiler type, number of API functions, number of custom functions, and entropy and checksum.

Flake [27] presented a method that constructs an isomorphism between the groups of functions that are used into two different versions of the same binary. He defined a graph centric analysis. This technique consists of representing an executable as a graph of graphs. The global structure is represented with a call graph, where each function represents a node. An edge exists between two function nodes f_i and f_j if and only if f_i calls f_j. Every function is represented with a Control Flow Graph (CFG). The CFG must have the following properties: a unique entry point and one or many nodes as an exit point (nodes that do not link to other nodes). In this work, the notion of structural matching is defined. It lies in matching the functions in two executable files by using the information generated from the call graph and different control flow graphs. In order to achieve the matching between two versions of the same executable, namely A and B, Flake considered a formal representation for each version. A program is formalized as follows:

$$A = \{\{a_1, a_2 \dots a_n\}, \{a_1^e, a_2^e \dots a_m^e\}\}$$
$$B = \{\{b_1, b_2 \dots b_l\}, \{b_1^e, b_2^e \dots b_k^e\}\}$$

The program is represented by the nodes and their edges (a_i^e and b_i^e are 2-tuple containing two nodes that are directly connected). Flake based his work on finding

iterative mappings between node sets. The iterative mapping is used due to the fact that the cardinality of the two sets is likely different. Each mapping considers two different subsets with the same cardinality. An initial mapping is created. It maps elements of a subset $A_1\{a_1, a_2 \dots a_n\}$ to $B_1\{b_1, b_2 \dots b_l\}$. This mapping is used to create iteratively a sequence of mappings.

In order to compare the control flow graphs, Flake considered a heuristic matching method. The initial step consists of finding isomorphism between two graphs. Once that is done, a 3-tuple $(\alpha_i, \beta_i, \gamma_i)$ is associated with each node: α_i represents the number of basic control blocks, β_i is the number of edges within a node, and γ_i is the number of edges originating from the node. The mapping is denoted by a function that maps a call graph to 3-tuple. The function is represented as follows:

$$s : C \to N^3$$

The inverse function is as follows:

$$s^{-1} : N^3 \to \varphi(\{c_1, \dots, c_o\})$$

This function retrieves the set of functions that map to a 3-tuple. The initial mapping is constructed by examining all 3-tuples that are generated from both program versions A and B. The functions a_i from A and b_j from B are mapped to each other if and only if they map the same 3-tuple and no other element in $\{a_1, a_2, \dots, a_n\}$ or $\{b_1, b_2, \dots, b_l\}$. The mapping is formalized as follows:

$$p_1(a_i) = b_j \Leftrightarrow |s^{-1}(s(a_i))| = 1 = |s^{-1}(s(b_j))| \wedge s(a_i) = s(b_j)$$

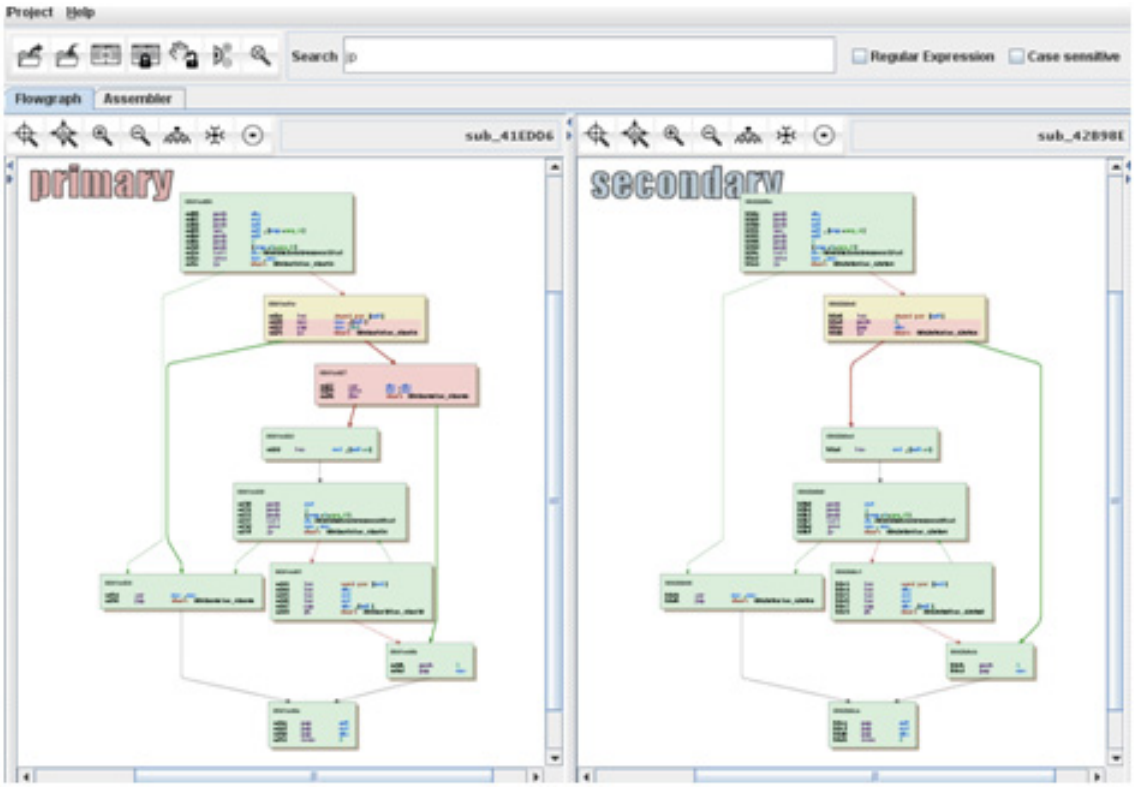
If the cardinalities of $s^{-1}(s(a_i))$ and $s^{-1}(s(b_j))$ are equal to one and a_i and b_j have the same tuple, p_1 maps a_i and b_j. Once the initial mapping is retrieved, improved mappings p_i are built iteratively by creating small subsets. The algorithm to generate mappings is as follows:

- Take the i^{th} element a_i from A_{i-1} and retrieve $p_{i-1}(a_i)$
- Let A'_i be the set of all functions a_k that have edges originating from a_i leading to a_k in φ and B'_i be the set of all functions b_o that have edges originating from $p_{i-1}(a_i)$ leading to b_o in φ.
- Construct $p'_i : A'_i \to B'_i$ in the same way p_1 was constructed
- $p_i(a_j) := p_{i-1}(a_j)$ if $a_j \in A_{i-1}$. If $a_j \notin A_{i-1}$ and the construction of p'_i yielded a match, $p_i(a_j) := p'_i(a_j)$. If the construction of p'_i did not yield a match and $a_j \notin A_{i-1}$ then $p_i(a_j)$ is undefined.

- A_i and B_i are the domain and image of p_i. The iteration carries on until all the nodes are browsed.

An implementation of the described method was developed as a plug-in for IDA Pro. A corroborated investigation has shown that the author is involved in the creation of the *BinDiff* tool.

BinDiff [3] is a tool that can compare binary files. It helps researchers find differences and similarities in disassembled code. It can also port symbols, comments, functions, and local names between disassembled code. BinDiff can be used to gather evidences for code theft or patent infringement. The following figure shows a screenshot of BinDiff. It shows the changed functions.



Based on the idea of BinDiff, VxClass [4] can also be used to compare disassembly code. VxClass can ignore byte-level changes such as instruction reordering or string obfuscation so that it can discover small changes in the code. The user can upload a sample of malware in a specific database. First, VxClass filters the sample malware by sorting out items the user already analyzed. Then, it finds out if that security incident under investigation is related to any previously analyzed item(s). Next,

it automatically removes the executable crypters from the code. Finally, VxClass compares the uploaded executable to the database of stored malware, and informs the user whether or not the binary is related to a piece of known malware.

Kruegel et al. [42] proposed a novel structural analysis technique to compare worm structures. This technique is based on comparison of colored graphs to characterize a worm's structure. The contributions of the paper are twofold: (1) The description of a fingerprinting technique based on CFG to detect similarities between variations of worms. (2) The fingerprinting technique is improved by a coloring scheme. The authors evaluated their prototype system to detect polymorphic worms. In this system, the authors claim that their technique can be adapted to network intrusion systems.

Anju et al. [8] proposed a method for malware detection based on control flow graph optimization. They claim that malware identification lies in syntactic as well as semantic features in binaries. They defined architecture for detecting malicious patterns in executable files. The architecture is broken into two components: Database Management and Program Analysis. Both components are based on disassembling executable binaries, optimizing assembly code, extracting the control flow graph, and optimizing the control flow graph.

Behavioral-Based Approach

Comparetti et al. [21] developed a system called *REANIMATOR* that allows the identification of dormant functionalities in malware. They exploit the fact that a dynamic malware analysis captures malware execution and reports its behavior. Their basic idea is to run a large set of malware in a sandbox environment and identify dynamically different malware functionalities. Once they are identified, the functionalities can be detected within new malware. The REANIMATOR system consists of three phases. The first two phases are responsible for generating a functionality-aware model for different behaviors. The last phase uses constructed models to check dormant behaviors.

Hybrid Approach

Wang et al. [60] presented a tool called BMAT that creates mappings between old and new versions of binaries. This tool is used to generate profile information for real-world applications and to illustrate how to propagate stale profiles from an old version

to a new version. In this work, the authors describe a binary matching algorithm of stale profile propagation.

Chapter 3

Problem Definition

First, an informal description of the assembly code clone detection problem is provided. Then, a formal problem description and an example are followed.

The problem of assembly code clone detection is informally described as follows: Given a large collection of previously analyzed assembly files and a specific target assembly file or a piece of target assembly code fragment, a user would like to identify all code fragments in the previously assembly files that are syntactically or semantically similar to the target assembly file or the piece of target assembly code fragment.

Let $A = \{A_1, \ldots, A_n\}$ be a collection of previously analyzed assembly files, where each assembly file A_f consists of m lines of assembly code, denoted by $f[1 : m]$. In the rest of this section, the assembly code has been assumed to be normalized, and the normalization process will be given in Section 4.1.3. A code fragment $f[a : b]$ in an assembly file A_i refers to a subsequence of assembly code from line a to line b in A_i inclusively, where $1 \leq a \leq b \leq m$. Let $|f[a : b]|$ denote the number of lines of assembly code in $f[a : b]$.

Two notions of clones are defined as follows. Intuitively, two code fragments are an *exact clone pair* if they have the same number of lines of assembly code and the same sequence of assembly code. Two code fragments that share similar instructions with respect to the mnemonics and operands are considered as an *inexact clone pair*.

Definition 3.0.1 (Exact clone) Let $f[a : b]$ and $f[c : d]$ be two arbitrary non-empty code fragments in A. $f[a : b]$ and $f[c : d]$ are an *exact clone pair* if $|f[a : b]| = |f[c : d]|$ and $f[a] = f[c], \ldots, f[b] = f[d]$. The relation $=$ denotes that two code fragments are identical with respect to the sequence of mnemonics and operands

appeared in the line of assembly code instruction. ■

Definition 3.0.2 (Inexact clone) Let $f[a:b]$ and $f[c:d]$ be two arbitrary non-empty code fragments in A. Let $sim(f[a:b], f[c:d])$ be a function that measures the similarity between two code fragments $f[a:b]$ and $f[c:d]$. $f[a:b]$ and $f[c:d]$ are an *inexact clone pair* if $sim(f[a:b], f[c:d]) \geq minS$, where $minS$ is a user-specified minimum similarity threshold $0 \leq minS \leq 1$. ■

Note that an exact clone pair has $sim(f[a:b], f[c:d]) = 1$. In other words, an exact clone pair is also an inexact clone pair with similarity equal to 1. Given a similarity threshold $minS$, the inexact clone detection process will also identify all exact clones. Thus, at first glance, the two notions of clones can be merged into one, and it seems to be unnecessary to develop two different clone detection processes for identifying exact and inexact clones separately. However, in real-life malware analysis, a reverse engineer sometimes wants to *efficiently* identify only the exact clones. The *problem of assembly code clone detection* is to identify *all* exact and inexact clones in a given collection of assembly code files A.

```
sub_76641161 proc near

21  mov     eax, [esi+8]
22  lea     edx, [eax+ebx*4+28h]
23  mov     eax, [eax+4]
24  push    edx
25  cmp     eax, 25h
26  jz      short loc_76641454
27  mov     ecx, [esi+10h]
28  jg      short loc_1000123E
29  mov     ecx, [esi+4]
30  push    ebp
31  mov     edx, esp
32  mov     eax, 2384h
33  call    ds:InitializeCriticalSection
34  call    __alloca_probe
35  push    esi
36  push    ebx
37  push    ecx
38  call    ds:_CT_MgmtALDisplayResult@8
39  mov     edx, [esi+4]
40  pop     edi
41  retn    4

sub_76641161 endp
```

Figure 11: Procedure *sub_*76641161

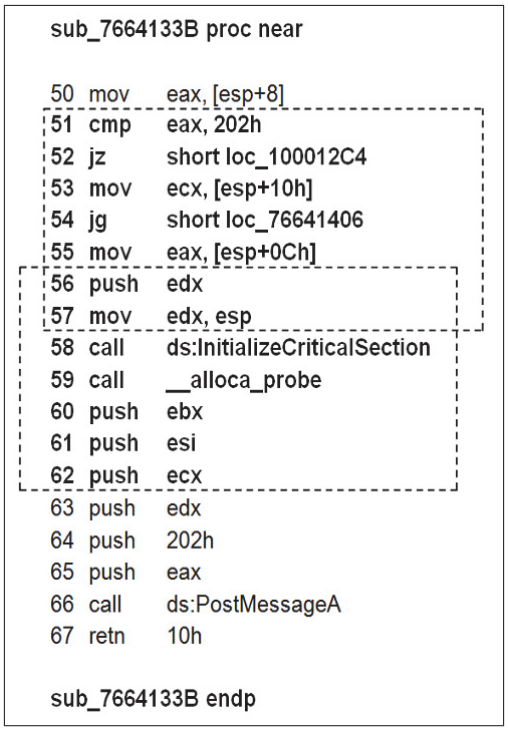


Figure 12: Procedure $sub\_7664133B$

Example 1 Suppose the collection of assembly code A contains only two procedures as shown in Figures 11 and 12. The code fragment $f[25, 31]$ in $sub\_76641161$ and the code fragment $f[51, 57]$ in $sub\_7664133B$ are an exact clone pair. Also, the code fragment $f[30, 36]$ in $sub\_76641161$ and the code fragment $f[56, 62]$ in $sub\_7664133B$ are an inexact clone pair. ■

Chapter 4

The Clone Detection System

The proposed clone detection system consists of five major components, namely pre-processor, clone region detector, clone merger, clone database, and clone visualizer. Figure 13 provides an overview of the implemented components. The pre-processor first disassembles a collection of binary files into assembly files. The clone region detector parses procedures in the assembly files, partitions each function into a sequence of regions, and identifies the clones among the regions. The output of the clone region detector is a collection of clone regions. Clone merger then merges the smaller clone regions into larger size clones. Then, the resulting clones are stored into a database, which is an XML file in our current implementation. Finally, the clone visualizer takes the XML file as input and interactively shows the clones to the user. A detailed description of each component is given below.

4.1 Pre-Processing

The pre-processor involves disassembling the binary code into assembly code, indexing the tokens from assembly code, and normalizing the assembly code for clone comparison.

4.1.1 Disassembler

This step disassembles the input binary files into assembly files A by using a disassembler, such as IDA Pro [2]. Each assembly file $A_f \in A$ contains a set of functions.

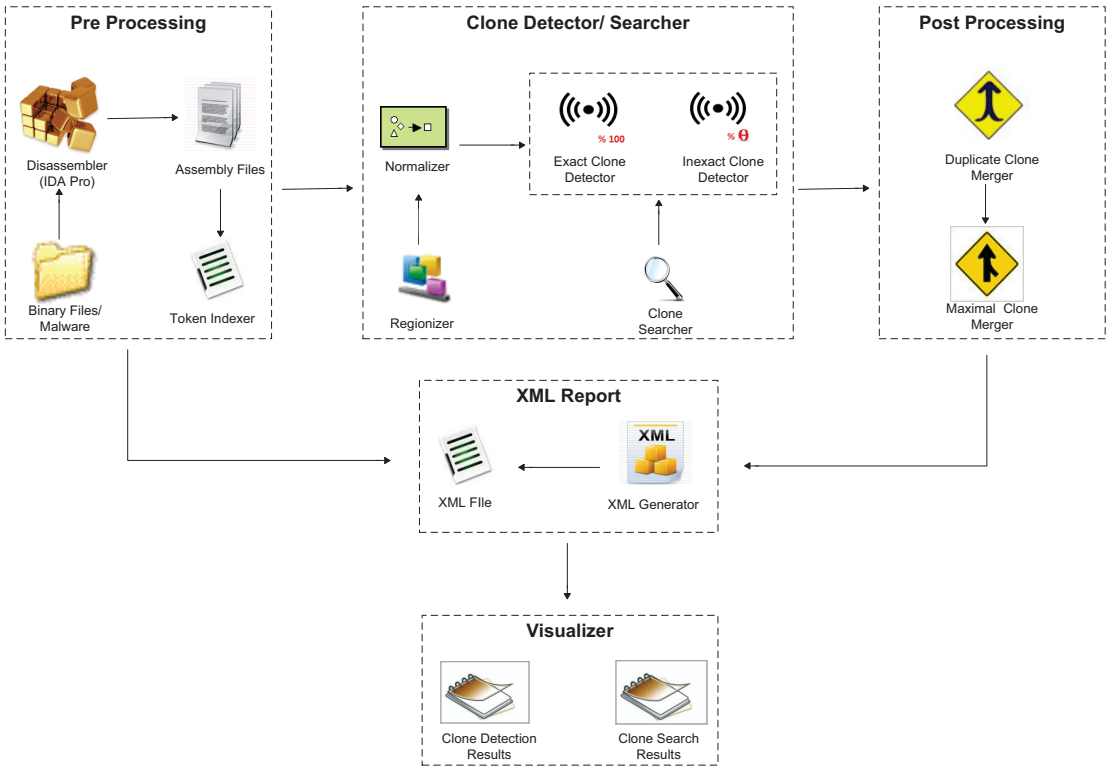


Figure 13: System Architecture

Each function contains a sequence of assembly instructions and each assembly instruction consists of a *mnemonic* and a sequence of *operands*. Mnemonics are used to represent the low-level machine operations. The operands can be classified into three categories, namely *memory reference*, *register reference* and *constant values*.

4.1.2 Token Indexer

In malware analysis, in addition to the ordinary clone detection process, a reverse engineer often wants to search for a specific token, such as a specific constant value. The objective of the token indexer is to parse the raw assembly files and create indexes for constants, strings, and imports, with the goal of facilitating direct access to the tokens. Specifically, the token indexer references tokens by their filenames and line numbers. These indexes are then stored into a user-specified XML file. See the XML file in Section 4.4 as an example. Figure 14 shows the search results on a string token "RpcTransServerFreeBuffer".

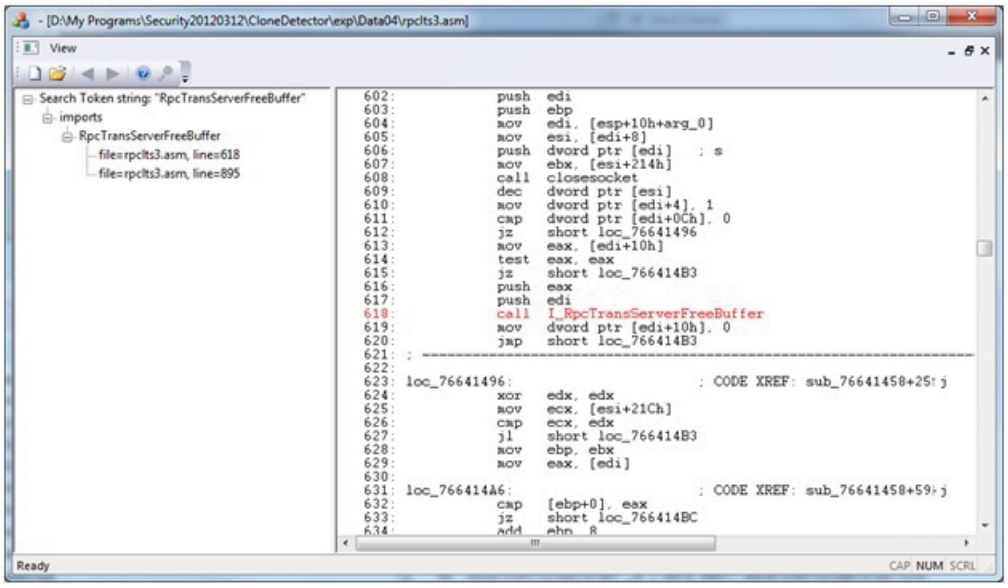


Figure 14: Search Capability (Search for String "RpcTransServerFreeBuffer")

4.1.3 Normalizer

Two code fragments may be considered as an exact clone even if some of their operands are different. For example, two instructions can be identical even if one uses the register `eax` and the other, `ebx`. Thus, it is essential that the assembly code is normalized before the comparison.

The objective of the normalizer is to generalize the *memory references*, *registers*, and *constant values* to an appropriate level chosen by the user. For constant values, the user has the flexibility to generalize them to VAL_x, where x is an index number, or to VAL, which simply ignores the exact constant value. . For memory references, the user has the flexibility to generalize them to MEM_x, where x is an index number, or MEM, which simply ignores the specific memory reference. For registers, the user has the flexibility to generalize them according to the normalization hierarchy depicted in Figure 15. The top-most level REG generalizes all registers disregarding their type. The next level differentiates between General Registers (e.g., EAX, EBX), Segment Registers (e.g., CS, DS), as well as Index and Pointer Registers (e.g., ESI, EDI). The third level breaks down the General Registers into 3 groups by size, namely 32-, 16-, and 8-bit registers. Finally, the bottom $REGx$ level appends a unique index to each distinct register based on their order of appearance in the code.

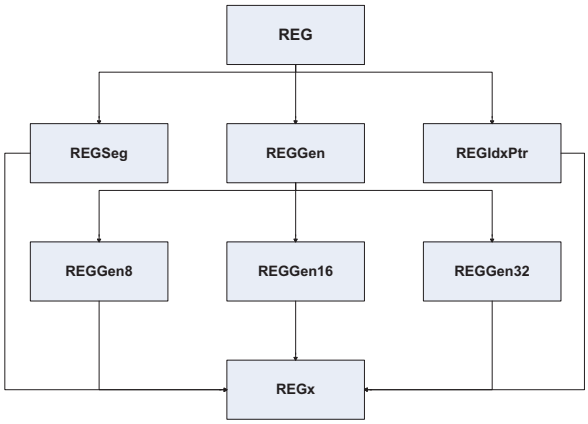


Figure 15: Normalization Hierarchy for Registers

Figure 16 illustrates the normalized tokens for different registers based on the hierarchy.

REG	
eax	REG
cs	REG
edi	REG
REGSeg, REGGen, REGIdxPtr	
eax	REGGen
cs	REGSeg
edi	REGIdxPtr
REGGen8, REGGen16, REGGen32	
eax	REGGen32
ax	REGGen16
ah	REGGen8
REGx	
eax	REG0
cs	REG1
edi	REG2

Figure 16: Register Normalization Example

Figures 17 and 18 show the normalized versions of $sub\_76641161$ and $sub\_7664133B$.

```
sub_76641161 proc near

21 mov   REGGen, MEM
22 lea   REGGen, MEM
23 mov   REGGen, MEM
24 push  REGGen
25 cmp   REGGen, VAL
26 jz    short loc_76641454
27 mov   REGGen, MEM
28 jg    short loc_1000123E
29 mov   REGGen, MEM
30 push  REGidxPtr
31 mov   REGGen, REGidxPtr
32 mov   REGGen, VAL
33 call  ds:InitializeCriticalSection
34 call  __alloca_probe
35 push  REGidxPtr
36 push  REGGen
37 push  REGGen
38 call  ds:_CT_MgmtALDisplayResult@8
39 mov   REGGen, MEM
40 pop   REGidxPtr
41 retn  VAL

sub_76641161 endp
```

Figure 17: Normalized *sub_76641161*

```
sub_7664133B proc near

50 mov   REGGen, MEM
51 cmp   REGGen, VAL
52 jz    short loc_100012C4
53 mov   REGGen, MEM
54 jg    short loc_76641406
55 mov   REGGen, MEM
56 push  REGGen
57 mov   REGGen, REGIdxPtr
58 call  ds:InitializeCriticalSection
59 call  __alloca_probe
60 push  REGGen
61 push  REGIdxPtr
62 push  REGGen
63 push  REGGen
64 push  VAL
65 push  REGGen
66 call  ds:PostMessageA
67 retn  VAL

sub_7664133B endp
```

Figure 18: Normalized *sub_7664133B*

4.2 Clone Detector / Searcher

The clone detector / searcher consists of four steps. The first step is to partition each function into an array of regions. The second and third steps are to identify the exact and inexact clones, respectively, among the regions created in the first step. Finally, the fourth step is to search a specific target code fragment through a repository of assembly files.

4.2.1 Regionizer

Each function is partitioned into an array of overlapping regions using a sliding window with a size of at most w statements and a step size s, where w and s are user-specified thresholds. Figure 19 shows the extracted regions of the normalized procedure *sub*_76641161 in Figure 17 with $w = 15$ and $s = 1$. Setting $s = 1$ ensures that no regions will be skipped, i.e., no clones will be missed.

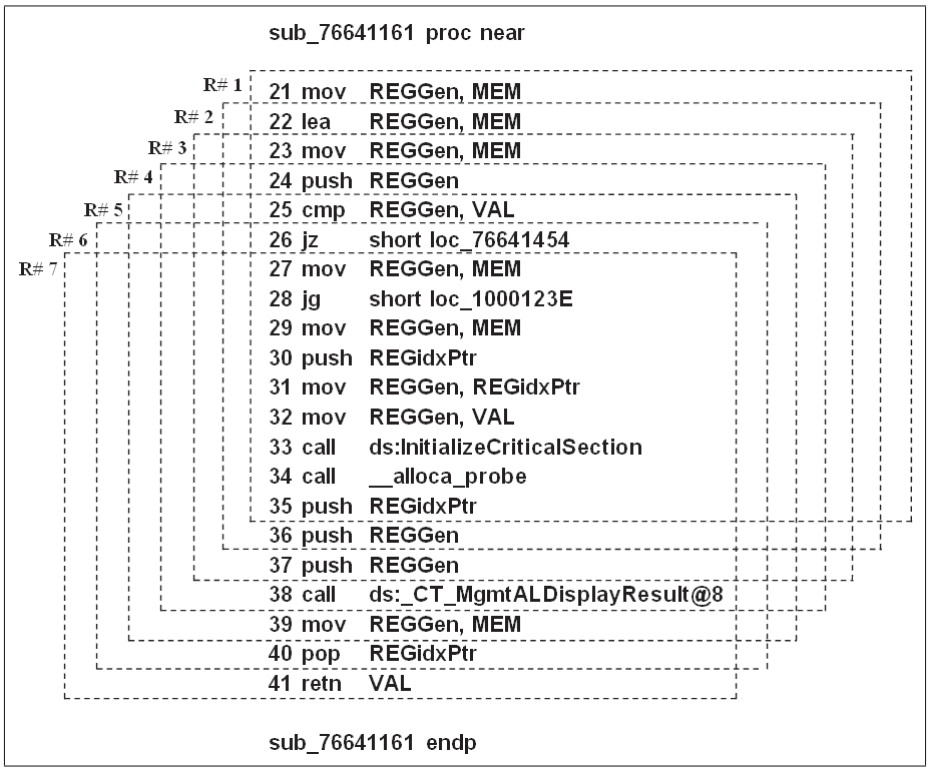


Figure 19: Regionization for $w = 15$ and $s = 1$

4.2.2 Exact Clone Detector

As mentioned in Definitions 3.0.1 and 3.0.2. A clone pair is defined as an unordered pair of code regions that have similar normalized statements. A clone cluster is a group of clone pairs. This step identifies the exact clone pairs among the regions by comparing their assembly code instructions. Two regions are considered as an exact clone pair if all the normalized statements in the two regions are identical. A naive approach to identify the exact clone pairs is to compare every pair of regions. Yet, this approach is too computationally expensive with complexity of $O(n^2)$, where n is the total number of regions. Thus, a hashing approach was employed. Specifically, two regions are considered as an exact clone pair if they share the same hash value. As this approach uses a hash algorithm to map each region to an integer value, all identical regions are mapped to the same bucket without false positives and false negatives. The process requires only one scan on the identified regions.

Algorithm 1 provides the details of the method. First, an empty hash table H is initialized. Each entry in the hash table contains a hash value v with a corresponding array of regions having such a hash value. In Lines 4-6, the method iterates through each region r, creates a hash value v, and adds the region r to the corresponding array $H(v)$. Each entry $H(v)$ contains an exact clone cluster. In Lines 7-9, the method iterates through each clone cluster and constructs an array of exact clone pairs denoted by EC.

4.2.3 Inexact Clone Detector

The objective of the inexact clone detector is to identify the inexact clone pairs from a given collection of regions. The detector first extracts some features from each region, constructs a feature vector, and then groups the feature vectors by similarity. Two regions are considered as an inexact clone pair if the similarity between their feature vectors is within a user-specified minimum similarity threshold.

The feature vectors are constructed based on five groups of features from the assembly instructions [50]. The first group of features includes all mnemonics. In other words, each distinct mnemonic forms a feature. The second group covers all operand types. The third group includes all combinations of mnemonics and the type of the first operand. The fourth group includes all combinations of the first two operands. Finally, the last group includes *maxOperands* number of distinct operands

Algorithm 1: Exact Clone Detector

```
input  : set of regions R
output: set of exact clone pairs EC
begin
    H ← ∅;
    EC ← ∅;
    foreach region r ∈ R do
        v ← hash(r);
        H(v) ← H(v) ∪ {r};
    foreach H(v) ∈ H do
        for i = 0 → |H(v)| do
            for j = i + 1 → |H(v)| do
                EC ← EC ∪ {(r_i, r_j)};
    return EC;
```

found in the code, where $maxOperands \geq 0$ is a user-specified threshold.

In this section, two inexact clone detection methods that iteratively improve the accuracy of clone detection are proposed in five steps.

Sliding Window Inexact Detection

Algorithm 2 provides an overview of the sliding window inexact detection method.

1. **Compute medians**: This step computes the median of each feature. The medians serve as a point of division for grouping the feature vectors in the subsequent steps. The feature values, however, may have a very large range. Therefore, the medians are computed to avoid the negative impact of outliers.

2. **Filter out features**: This step filters out the features that have their median equal to 0. The rationale is that some features may appear only once or a few times in all extracted regions, implying that they are unimportant for the purpose of region comparison. Thus, removing the features with a median of zero can improve the accuracy and efficiency of the inexact clone detection method.

3. **Generate binary vectors**: This step constructs a binary vector for each region by comparing the feature vector of the region with the median vector. If a feature value is larger than the corresponding median, then 1 is inserted into

the entry of the binary vector. Otherwise, 0 is inserted.

4. **Partition into sub-vectors**: The fourth step is to partition each binary vector into a sequence of sub-vectors using a sliding window of size $SBSize$.

5. **Hash sub-vectors**: Given that the size of each sub-vector is $SBSize$, there are 2^{SBSize} possible combinations of binary values. For each sub-vector, a hash with 2^{SBSize} number of buckets is created to store the regions having the same sub-vector values. The regions are hashed by computing the decimal number of the sub-vector values. The inexact clone pairs are identified in this step by keeping track of the frequency of region co-occurrences in all inexact hash tables' buckets. The region pairs with the number of co-occurrences above or equal to the similarity threshold $minS$ are considered as inexact clone pairs.

Example 2 Figure 20 shows a collection of features generated from a dataset after the filtering process. For simplicity, just a small set of features are shown here. The dashed rectangles show the sub-vectors of the feature vector with a user-defined sub-vector of size 5. There are $n - 5 + 1$ sub-vectors for n extracted features after the filtering process and $2^5 = 32$ possible hash values (decimal numbers) for each sub-vector that makes the size of each associated inexact hash table 32. Step 5 maps the regions into these hash tables by computing the decimal number of their binary vectors. ■

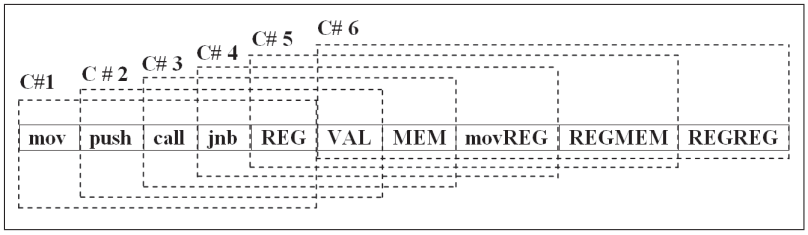


Figure 20: Step 3 - Sliding Window Inexact Detection Method with $SBSize = 5$

Two-Combination Inexact Detection

This method follows the same general steps, but the detailed process in step 3 is different. Instead of creating sub-vectors with the user-defined length sliding windows, all possible two-combination of the remaining features after the filtering process are constructed. Each two-combination vector acts as a sub-vector. Then, each feature vector

Algorithm 2: Inexact Clone Detector: Sliding Window Method

input : set of regions R
set of features F
similarity threshold $minS$
output: set of inexact clone pairs IC

```
begin
    SBSize ← sub − vectors′size; Binary ← ∅;
    H_k ← ∅;
    M ← ComputeMedians(F);                                   /* Step1 */
    foreach m ∈ M do                                         /* Step2 */
        if m = 0 then
            M ← M − m;
    foreach r ∈ R do                                         /* Step3 */
        for k = 0 → length(M) do
            if F[k] ≥ M[k] then
                Binary[k] ← 1;
            else
                Binary[k] ← 0;
    foreach r ∈ R do                                         /* Step4 */
        for i = 0 → length(Binary) − SBSize + 1 do
            for j = 0 → SBSize do
                sub − vector_i[j] ← Binary[i + j];
    foreach r ∈ R do                                         /* Step5 */
        foreach k = 0 → Numberofsub − vectors do
            H_k ← Compute the decimal number;
            R′ ← find other regions with the same hash value
        foreach r′ ∈ R′ do
            if r and r′ occurred more than the minS threshold then
                IC ← IC ∪ {(r, r′)};
    return IC;
```

is mapped into its sub-vectors. Sub-vectors have the same size as two-combination which is equal to 2. In this case, the user does not have the flexibility to choose the size of sub-vectors.

Example 3 Figure 21 shows all possible two-combinations of the features in Figure 20, each of which is a sub-vector. Let n be the number of features after the filtering process. There are $C(n,2) = \frac{n\times(n-1)}{2}$ sub-vectors. Given that each sub-vector is a binary vector of size 2, there are $2^2 = 4$ possible hash values, implying that each inexact hash table contains 4 entries. This method maps the regions into sub-vectors based on their binary vectors generated from Step 3. ■

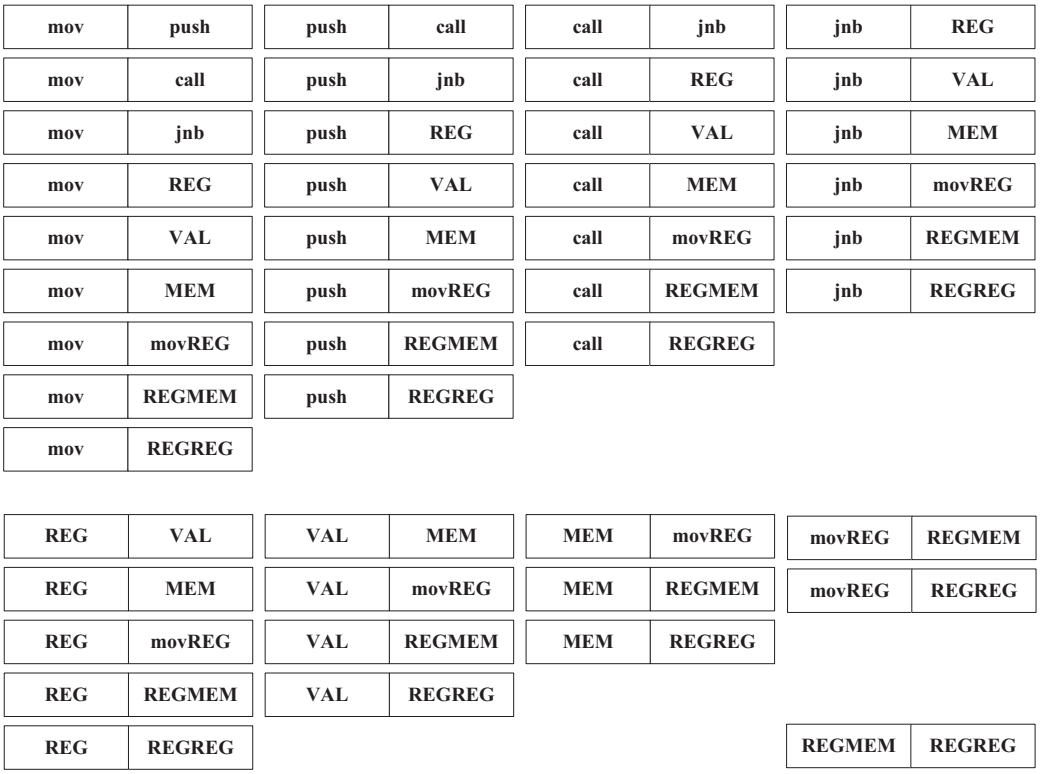


Figure 21: Step 3 - Two-Combination Inexact Detection Method

The two proposed inexact detection methods generate different numbers of sub-vectors and different sub-vector sizes. These characteristics affect the efficiency and scalability of the program.

The sliding window method considers only the sub-vectors with consecutive features, while the two-combination method considers all possible two combinations. Therefore, the set of sub-vectors generated by the sliding window method is a subset

of the sub-vectors generated by the two-combination method. As a result, the sliding window method performs better than the two-combination method in terms of scalability, but the two-combination method performs better in terms of recall rate. Experimental results also support this observation.

Comparing with LSH Approach

Sæbjørnsen et al. [50] presented an inexact clone detection method to identify inexact clone pairs by using the locality-sensitive hashing (LSH) to find the nearest neighbor vectors of a given query vector. Their assumption on uniform distribution of vectors in LSH method affects the number of false-negative errors, i.e., the recall rate. LSH consists of m hash functions. Each hash function h_i maps a vector v to a binary vector by computing the dot product of v and a base vector b_i. If the computed result is negative, the vector will be mapped to 0. Otherwise, it will map to 1. The base vector and vector v must share the same size. Using these parameters, the LSH value $lsh(v)$ of a vector v is defined as the following equation:

$$lsh(v) = (h_1(v), h_2(v), \ldots, h_m(v)) \tag{1}$$

In brief, the LSH method splits a vector space into 2^m sub-spaces by m base vectors. These base vectors are chosen randomly and the distribution of vectors are not considered. If the distribution of vectors is lopsided, then LSH cannot split the vector space efficiently, resulting in incorrect subspace assignment for some vectors. The accuracy of finding the nearest neighbor problem using LSH depends on parameters selection which is challenging in large dimension feature vectors. Also, due to the employment of randomization, the clone results produced by LSH are non-deterministic. Some malware analysts clearly indicate that this non-deterministic behaviour is unacceptable, as it will be very difficult for the reverse engineers to produce a consistent analysis on malware. To avoid the non-deterministic behaviour as in LSH, the proposed methods employ fixed parameters derived from the data. The first one is the number of subspaces which is the number of sub-vectors and the second parameter is the sub-spaces dimensions.

4.2.4 Clone Searcher

Given a target code fragment, the *Clone Searcher* module uses the previously described detection methods to search for the *exact* and *inexact* matching clones in a collection of previously analyzed assembly files.

4.3 Post-Processing

4.3.1 Duplicate Clone Merger

The inexact clone detector may misclassify two consecutive regions to be a clone. This step is to remove the clones that are highly overlapping consecutive regions. This happens when the stride s is smaller than the windows size w. A user-specified maximum overlapping threshold $maxO$ is defined that indicates the fraction of allowed overlapped instructions of two consecutive regions which can still be considered as a clone pair.

Example 4 Figure 22 provides an example for this step. It shows two consecutive regions with a window of size 10 and an overlapping ratio of 0.6. Suppose $maxO = 0.5$. Since the overlapping ratio is above $maxO$, the clone pair is discarded. ■

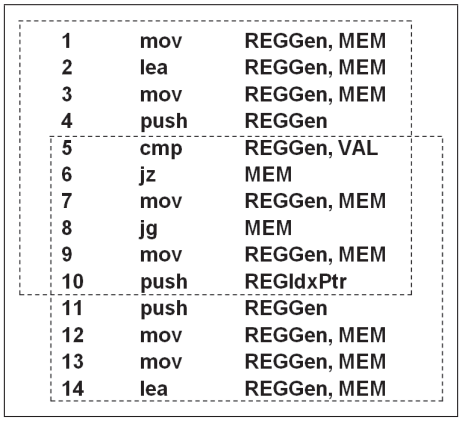


Figure 22: Duplicate Clone Merger with $w = 10$, $s = 4$, and an Overlapped Size of 0.6

4.3.2 Maximal Clone Merger

Since the clone detection processes operate on regions, the size of the identified clones is bounded by the window size w. As a result, a natural large clone fragment may

be broken down into small, consecutive cloned regions, making the analysis difficult. The objective of this step is to merge the smaller consecutive clone regions into a larger clone. All identified clone fragments are then stored in the user-specified XML file.

Algorithm 3 provides the maximal clone merger design where CP is the set of identified clone pairs and MC is the set of maximal merged clone pairs after the merging process. The *overlap* function finds the overlapped clones in lines 4-5. Suppose clone pairs c and c' are a pair of regions $\{A, B\}$ and $\{A', B'\}$ respectively. Two clones are overlapped if each of their regions shares some instructions. Hence, c and c' are overlapped cloned pairs if $\{A, A'\}$ and $\{B, B'\}$ have overlapped instructions.

Algorithm 3: Maximal Clone Merger

```
input  : set of clone pairs CP
output: set of maximal merged clone pairs MC
begin
    MC ← ∅;
    foreach clone pair c and c′ ∈ CP do
        if overlap(region A ∈ c, region A′ ∈ c′) & overlap(region B
        ∈ c, region B′ ∈ c′) then
            CP ← merge(c, c′);
    MC ← CP;
    return MC;
```

Example 5 Figure 23 provides an example on the maximal clone merge. With window size $w = 5$, every region in lines 50 - 60 on the left corresponds to a region in lines 105 - 115 on the right, represented by 6 clone pairs. Since all 6 clone pairs are consecutive, they are merged into one clone pair as indicated by the dashed rectangles. ■

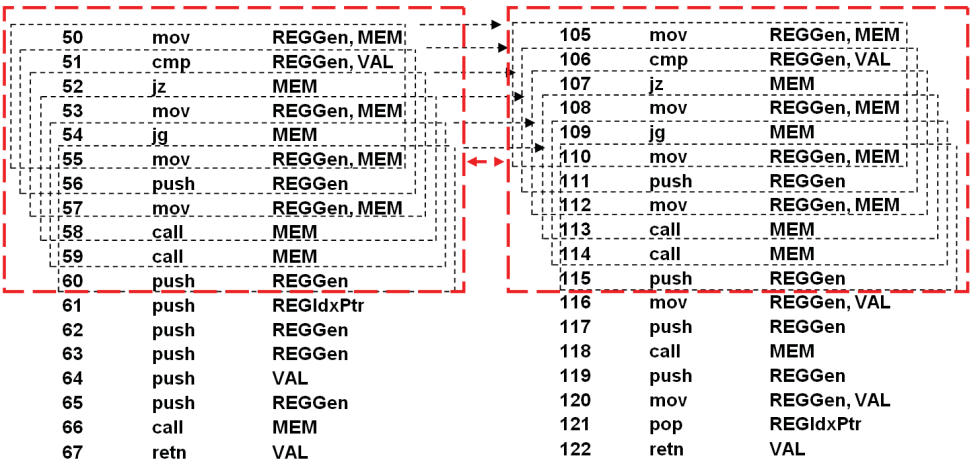


Figure 23: Maximal Clone Merger with $w = 5$ and $s = 1$

4.4 XML Output

The clone detection results are stored in an XML file. The XML file contains four nodes, namely *parameters*, *assembly_files*, *clone_files*, and *token_references*. Refer to Figure 24 as an example.

Figure 24: Sample XML File

- The *parameters* node stores the user-specified parameters, such as the window size w, step size s, minimal similarity threshold $minS$, maximal overlapping threshold $maxO$, normalization level and maximum number of distinct operands $maxOperands$.

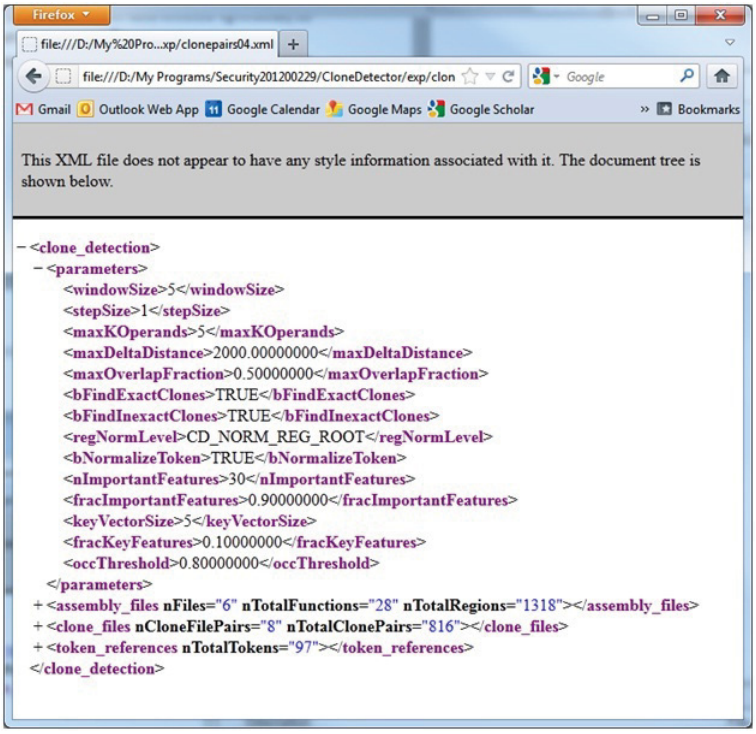


Figure 25: Sample XML File (parameters)

- The *assembly_files* node stores a list of assembly files. The primary objective is to assign a unique *fileID* to each assembly file for subsequent references. Some basic statistics, such as the number of functions and the number of regions found, are also stored in the corresponding node. Refer to Figure 26 as an example.

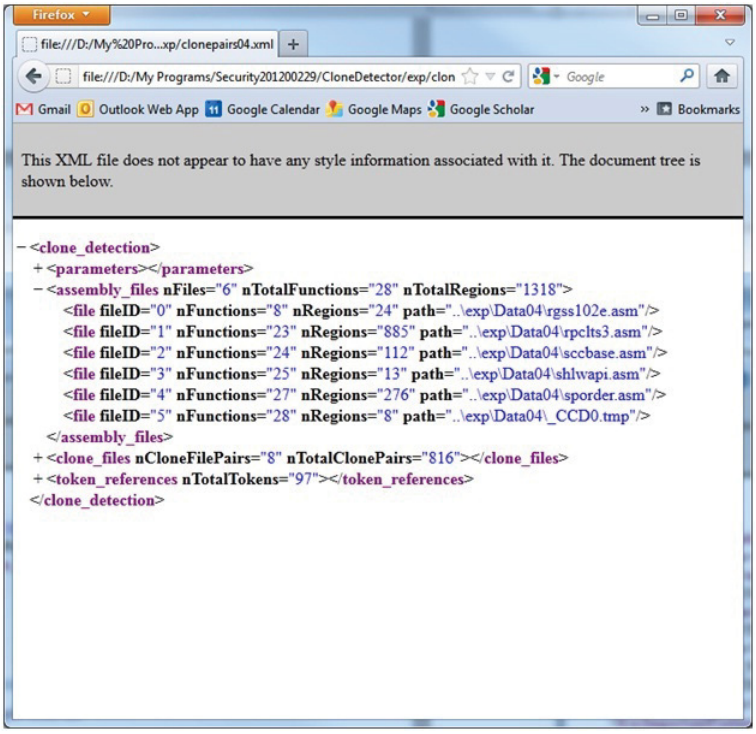


Figure 26: Sample XML File (assembly_files)

- The *clone_files* node stores the clone search results. Specifically, the *clones_files* node stores a list of *clone_files*, in which each *clone_file* stores a list of *clone_pairs*. Each *clone_pair* stores the location references for the clone pair. For example, Figure 27 contains 8 pairs of clone files. File with *fileID = 5* and File with *fileID = 1* share 4 clone pairs. The first entry, for example, indicates that lines 3-7 in *fileID = 5* is a clone of lines 600-604 in *fileID = 1*.

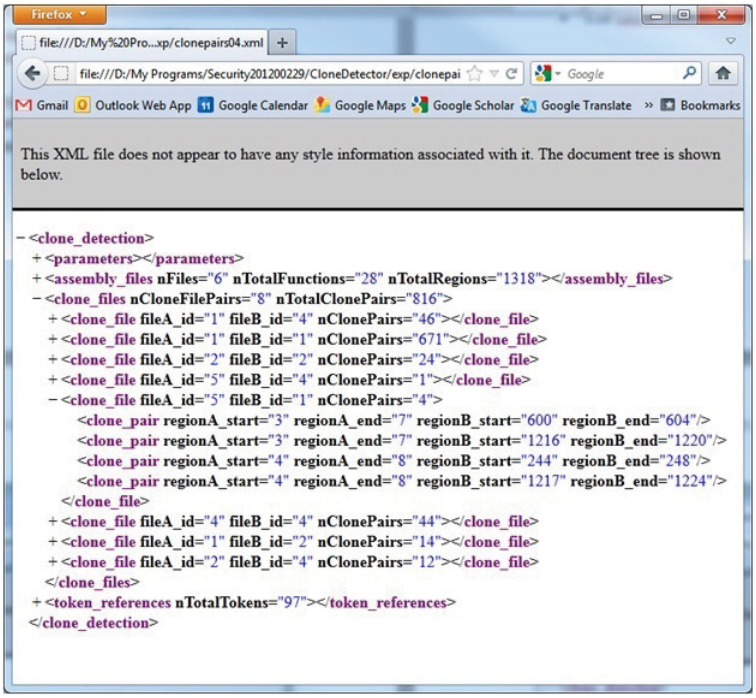


Figure 27: Sample XML File (clone_files)

- The *token_references* node stores the token search results. Specifically, the *token_references* node stores three lists of tokens, namely constants, strings, and imports. For example, Figure 28 contains 97 tokens. Specifically, the string token *PackedCatalogItem* appears twice in line 388 and line 504 in file with *fileID = 4*.

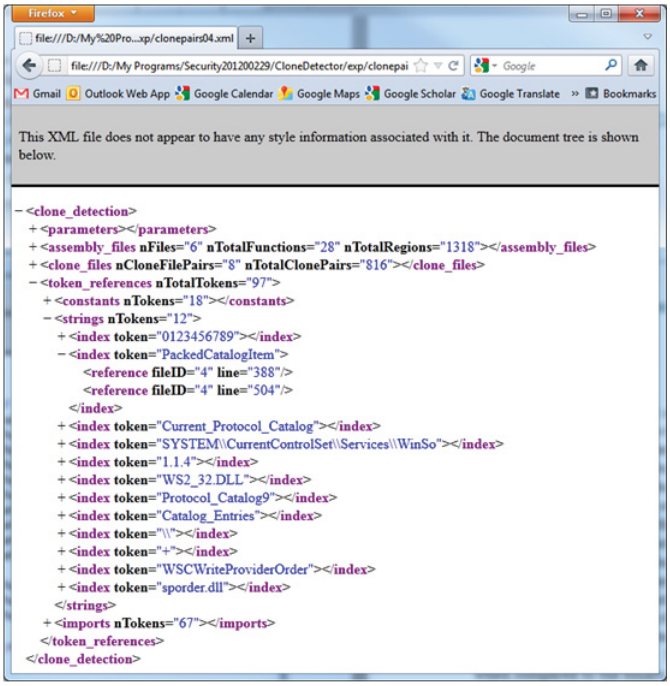


Figure 28: Sample XML File (token_references)

4.5 Visualizer

A graphical user interface (GUI) has been implemented to allow the user to input the required parameters, read the user-specified target code fragment or target tokens, and interactively identify the matched clone fragments or tokens from the assembly files. First, the user has to specify the set of parameters as shown in Figure 29 and Figure 30. This set consists of *assembly folder path*, *XML report destination path*, *window size*, *step size*, maximal overlapping threshold $maxO$, minimal similarity $minS$, $maxOperands$, *register normalization level*, *choice of inexact detection method*, and *sub-vector size*, if the inexact detection is checked as sliding window method.

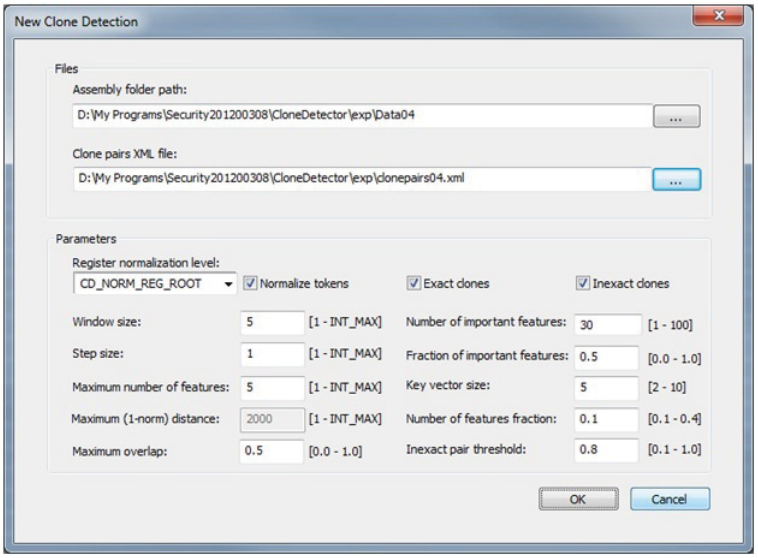


Figure 29: Clone Detection (Input Parameters)

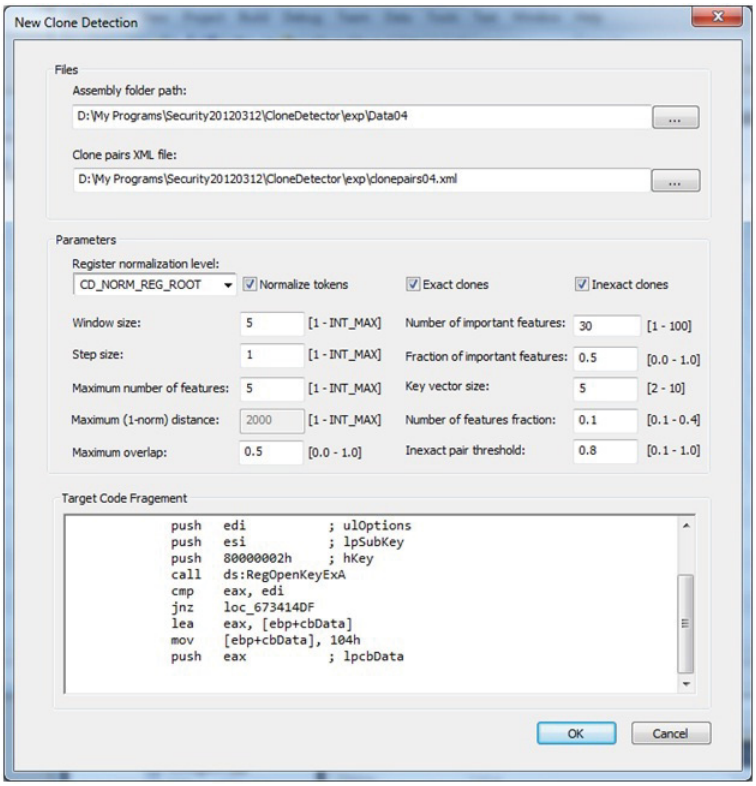


Figure 30: Clone Search

In this particular example, the program parses the assembly files form *Assembly folder path* and stores the clone results in *clonepairs04.xml*, which shows all the clone files in Figure 31.

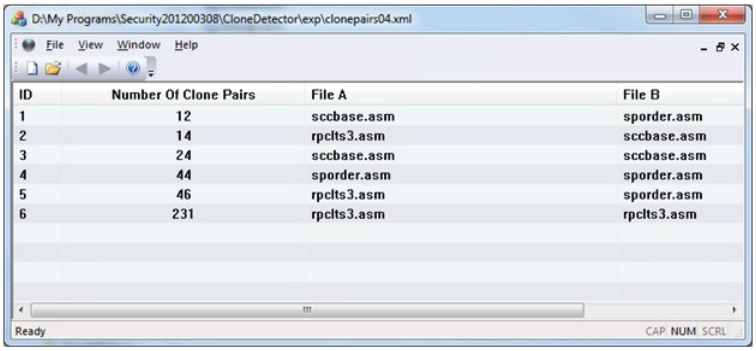


Figure 31: GUI (Clone File Pairs Found)

Next, the user clicks on a pair of clone files, say the second one with *rpclts3.asm* and *sccbase.asm*. Then, a list of clone pairs in these two files is shown in Figure 32. The cloned fragments in the two files are then highlighted in red.

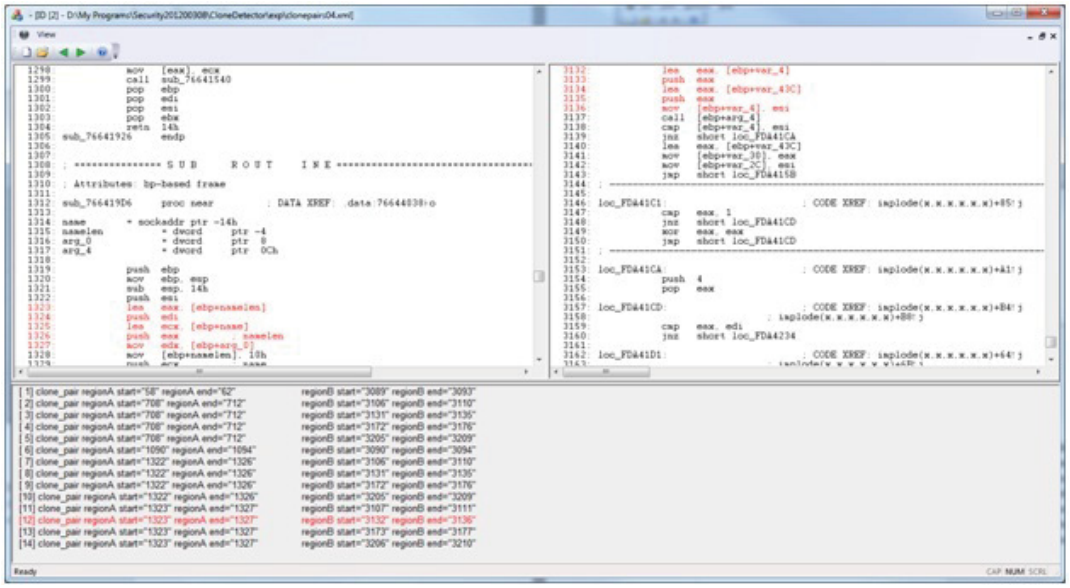


Figure 32: GUI (Code Fragment of Clone Pair)

Chapter 5

Experimental Results

The objective of the empirical study is to evaluate the proposed assembly code clone detection method in terms of accuracy, efficiency, and scalability. All experiments were performed on an Intel Xeon X5460 3.16 GHz Quad-Core processor-based server with 48GB of RAM running Windows Server 2003.

The experiments were conducted on three sets of binary files. The first dataset is an assortment of DLL files from an Operating System converted into 18 assembly files by IDA Pro [2]. The second dataset contains two well-known malware, *Zeus* and *Blaster*. *Zeus* is a trojan horse that extracts banking information using man-in-the-browser keystroke logging and form grabbing. *Blaster* is a computer worm that spreads by exploiting a buffer overflow on computers running Microsoft Windows simply by spamming itself to large numbers of random IP addresses. Table 2 shows some basic information on the two malware disassembled by IDA Pro [2]. The third dataset is an assortment of 70 malware obtained from the NNational Cyber-Forensics and Training Alliance (NCFTA) Canada [5]. The files were disassembled using IDA Pro [2]. The total size of these file these are more than 10 MB.

Name	Type	Size	# of Functions	# of LOC
Zeus	Trojan Horse	9 MB	45954	594153
Blaster	Worm	70 KB	13	2642

Table 2: Malware Specifications

5.1 Accuracy

To evaluate the accuracy of the proposed clone detection methods, some code fragments were first selected from the 18 assembly files. Then, clones of the code fragments were manually identified in the assembly files. Finally, the detection results of the proposed system were compared with the manually identified clones in order to compute the following three objective measures:

$$Precision(Solution, Result) = \frac{n_{ij}}{|Result|} \tag{2}$$

$$Recall(Solution, Result) = \frac{n_{ij}}{|Solution|} \tag{3}$$

$$F(Solution, Result) = \frac{2 \times Recall \times Precision}{Recall + Precision} \tag{4}$$

where *Solution* is the set of manually identified clone fragments, *Result* is the set of code fragments in a clone detection result, and n_{ij} is the number of code fragments in both *Solution* and *Result*. Intuitively, *F(Solution, Result)* measures the quality of the clone detection *Result* with respect to the *Solution* by the harmonic mean of *Recall* and *Precision*. As the goal is to evaluate the quality of the results with respect to a manually identified solution, it is infeasible to perform the evaluation in this manner on an extremely large collection of assembly files.

For the assortment of DLL files, Figure 33 shows the precision, recall, and F-score for step size $s = 1$, maximum number of distinct features $maxOperands = 40$, minimum similarity threshold $minS = 0.5$ and $minS = 0.8$. The value of s is chosen to be one to consider all possible regions in the datasets. Both the sliding window and the two-combination inexact detection methods are evaluated. Experimental results show a better precision for sliding window inexact detection when compared with the two-combination method. By considering the fact that sliding window inexact detection has less number of sub-vectors, Figure 33 implies that a higher number of inexact sub-vectors increases the number of false positives. In contrast, the two-combination method yields a higher recall rate due to the reason explained in Section 4.2.3. The precision and F-score are consistently above 75% for both inexact detection methods. The recall is above 80% for the sliding window and 100% for the two-combination method, suggesting that the clone detection methods are effective.

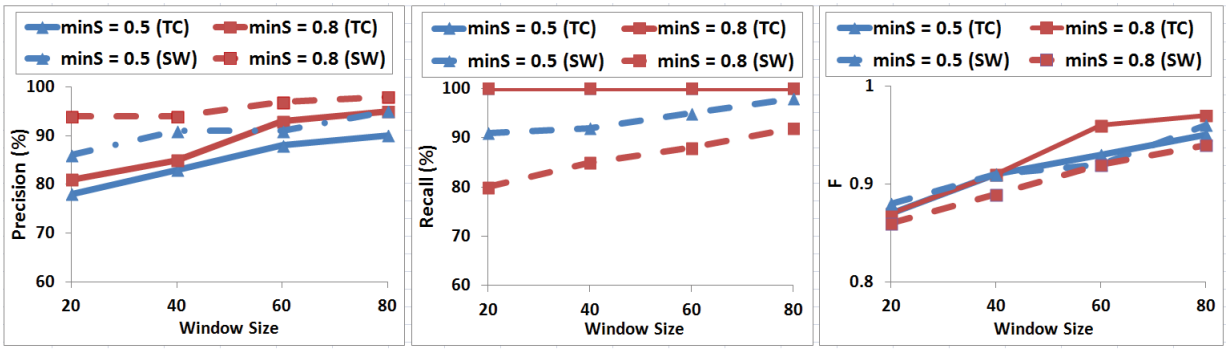


Figure 33: Accuracy (DLL files) with $s = 1$ and $maxOperands = 40$

To evaluate the precision of the second dataset (*Zeus* and *Blaster*), the first 10 regions were selected from each of the malware. For each selected region, the proposed clone detection methods were used to search for its clones in the rest of the code. Then, each of the identified clone in the result was manually examined in order to compute the precision. With step size $s = 1$, maximum number of distinct features $maxOperands = 40$, minimum similarity threshold $minS = 0.8$, and window size w ranging from 20 to 80, the precision consistently stayed above 90%, suggesting that the proposed methods are effective in identifying clones in malware.

Window Size:	20	40	60	80
# of Exact Clones	18010	17225	17162	16971
# of Inexact Clones (SW)	266335	272008	274346	759953
# of Inexact Clones (TC)	285132	441575	736396	1053801

Table 3: Number of Clones (Malware Assortment)

The number of exact and inexact clones identified by the proposed methods in the third dataset (malware collection) was also evaluated for different window sizes for both the sliding window and two-combination inexact detection methods. Table 3 shows that there is a large number of exact and inexact clones in malware. The result suggests that malware programmers share many codes at both regional and functional levels. Also, the experimental results suggest that the two-combination method (TC) can identify more clones than the sliding window method.

5.2 Efficiency

Figure 34 depicts the runtime for both exact and inexact clone detection methods for a window size ranging from 20 to 80 for the malware assortment dataset. The process took 26 to 30 seconds in total when the sliding window inexact clone detection method is used and 41 to 68 seconds in total for the two-combination inexact clone detection. In general, the runtime decreases as the window size increases, because fewer number of regions results in fewer number of clones.

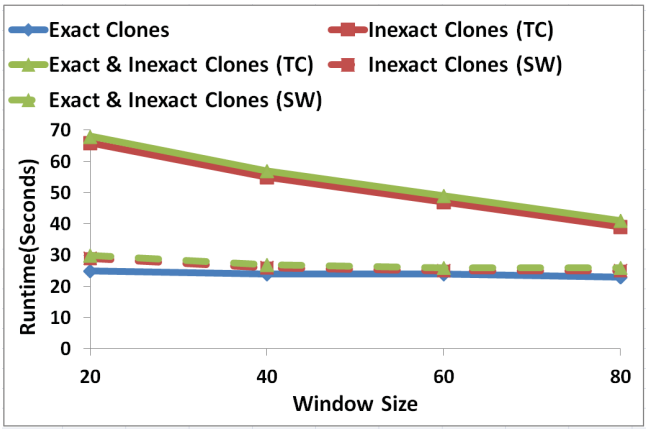


Figure 34: Runtime vs. Window Size (Malware Assortment)

5.3 Scalability

Figure 36 illustrates the runtime, in seconds, of each step for 10 to 70 malware files with window size $w = 40$, step size $s = 1$, maximum number of distinct features $maxOperands = 40$, and minimum similarity threshold $minS = 0.8$, for the two-combination inexact detection method. The total processing time for the sample malware assortment ranges from 35 to 980 seconds. Figure 35 shows the runtime using the same dataset and settings for the sliding window inexact detection method. In this case, the total processing time ranges from 8 to 258 seconds. As mentioned in Section 4.2.3, the sliding window inexact clone detection method performs better in terms of scalability, as it has fewer number of inexact sub-vectors.

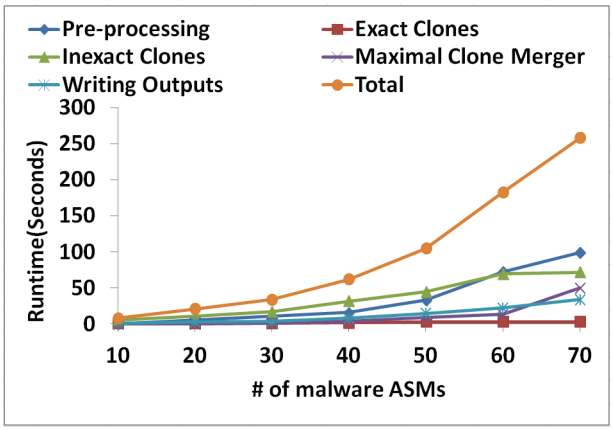


Figure 35: Scalability (with Sliding Window)

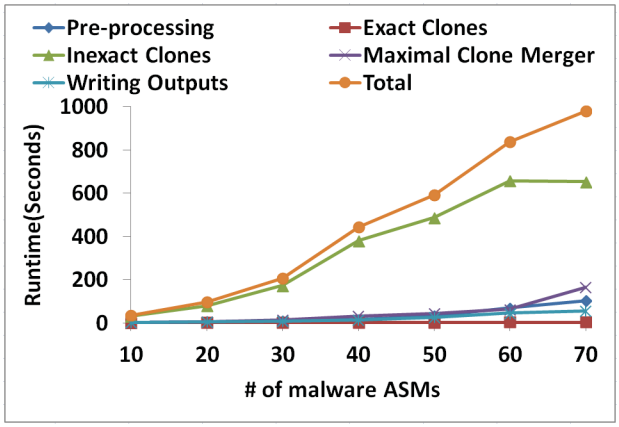


Figure 36: Scalability (with Two-Combination)

Chapter 6

Conclusion and Future Work

In this thesis, we reviewed the source code clone detection techniques as well as the feasibility of applying them to assembly code clone detection. Then, a metric-based assembly code clone detection system was presented with novel ideas capable of finding both exact and inexact clones. To evaluate the system, a comprehensive experiment was conducted on real-life binary files obtained from an Operating System and malware obtained from NCFTA Canada [5]. Experimental results suggest that the implemented system can effectively identify exact and inexact clones in assembly code.

The contributions of this thesis are summarized as follows. First, two efficient and effective inexact clone detection method capable of finding type III clones are proposed. Experimental results suggest that the two-combination inexact detection method can eliminate all false negatives. Second, unlike the LSH approach employed in Sæbjørnsen et al.'s work [50], our proposed clone detection methods are deterministic, which is an important property for malware analysis as specified by analysts. Third, a flexible normalization scheme is implemented to normalize assembly instruction so that clone detection can be performed at different levels depending on the purpose of clone detection to find type I and type II clones. Fourth, the capability of searching code fragments through a code repository is added. Finally, a graphical user interface is implemented to let users browse the identified clones.

The current implementation is a prototype system that evaluates the feasibility of assembly code clone detection. The system can be further improved in the following two directions.

1. *Inexact clone detection*: Though the currently implemented inexact clone detection produces reasonably high-quality results, the quality of the clone detection process is sensitive to the user-specified parameters. One potential improvement is to replace the hash-based approach by a cluster-based data mining approach.

2. *Scalable implementation*: In the current implementation, all computations are performed in memory. To build a real-life clone detection system, the one must develop a disk-resident version of the algorithm that stores the feature vectors into a database, instead of regenerating the feature vectors every time.

Most of the works in the literature, including the method implemented in this thesis, focus on identifying syntactic clones. Yet, a real research challenge is to identify the semantic clones that are syntactically different. This remains a challenging research problem in the area of source code and assembly code clone detection.

Bibliography

[1] Re-Google. http://regoogle.carnivore.it.

[2] IDA Pro. http://www.hex-rays.com/products/ida.

[3] BinDiff. http://www.zynamics.com/bindiff.html.

[4] VxClass. http://www.zynamics.com/vxclass.html.

[5] National Cyber-Forensics and Training Alliance CANADA (NCFTA). http://www.ncfta.ca.

[6] M.I. Abouelhoda, S. Kurtz, and E. Ohlebusch. Replacing suffix trees with enhanced suffix arrays. *Journal of Discrete Algorithms*, 2(1):53–86, 2004.

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

[8] S.S. Anju, P. Harmya, N. Jagadeesh, and R. Darsana. Malware detection using assembly code and control flow graph optimization. In *Proceedings of the 1st Amrita ACM-W Celebration on Women in Computing in India*, page 65. ACM, 2010.

[9] B.S. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, pages 49–49, 1993.

[10] B.S. Baker. On finding duplication and near-duplication in large software systems. In *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on*, pages 86–95. IEEE, 1995.

[11] M. Balazinska, E. Merlo, M. Dagenais, B. Lague, and K. Kontogiannis. Measuring clone based reengineering opportunities. In *Software Metrics Symposium, 1999. Proceedings. Sixth International*, pages 292–303. IEEE, 1999.

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

[13] I.D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. Clone detection using abstract syntax trees. In *Software Maintenance, 1998. Proceedings. International Conference on*, pages 368–377. IEEE, 1998.

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

[15] I. Briones and A. Gomez. Graphs, entropy and grid computing: Automatic comparison of malware. *Virus Bulletin*, 2008.

[16] R. Brixtel, M. Fontaine, B. Lesner, C. Bazin, and R. Robbes. Language-independent clone detection applied to plagiarism detection. In *Source Code Analysis and Manipulation (SCAM), 2010 10th IEEE Working Conference on*, pages 77–86. IEEE, 2010.

[17] D. Bruschi, L. Martignoni, and M. Monga. Code normalization for self-mutating malware. *Security & Privacy, IEEE*, 5(2):46–54, 2007.

[18] E. Burd and J. Bailey. Evaluating clone detection tools for use during preventative maintenance. In *Source Code Analysis and Manipulation, 2002. Proceedings. Second IEEE International Workshop on*, pages 36–43. IEEE, 2002.

[19] E. Carrera and G. Erdélyi. Digital genome mapping–advanced binary malware analysis. In *Virus Bulletin Conference*, pages 187–197, 2004.

[20] R. Cilibrasi and P.M.B. Vitányi. Clustering by compression. *Information Theory, IEEE Transactions on*, 51(4):1523–1545, 2005.

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

[22] I.J. Davis and M.W. Godfrey. Clone detection by exploiting assembler. In *Proceedings of the 4th International Workshop on Software Clones*, pages 77–78. ACM, 2010.

[23] S. Ducasse, O. Nierstrasz, and M. Rieger. On the effectiveness of clone detection by string matching. *Journal of Software Maintenance and Evolution: Research and Practice*, 18(1):37–58, 2006.

[24] T. Dullien, E. Carrera, S.M. Eppler, and S. Porst. Automated attacker correlation for malicious code. Technical report, DTIC Document, 2010.

[25] S. Dumais et al. Latent semantic indexing (lsi) and trec-2. *NIST SPECIAL PUBLICATION SP*, pages 105–105, 1994.

[26] W.S. Evans, C.W. Fraser, and F. Ma. Clone detection via structural abstraction. *Software Quality Journal*, 17(4):309–330, 2009.

[27] H. Flake. Structural comparison of executable objects. In *Proc. of the International GI Workshop on Detection of Intrusions and Malware & Vulnerability Assessment, number P-46 in Lecture Notes in Informatics*, pages 161–174, 2004.

[28] B. Hummel, E. Juergens, L. Heinemann, and M. Conradt. Index-based code clone detection: incremental, distributed, scalable. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–9. IEEE, 2010.

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

[30] J. Jang, D. Brumley, and S. Venkataraman. Bitshred: Fast, scalable malware triage. *Cylab, Carnegie Mellon University, Pittsburgh, PA, Technical Report CMU-Cylab-10-022*, 2010.

[31] J.H. Ji, S.H. Park, G. Woo, and H.G. Cho. Source code similarity detection using adaptive local alignment of keywords. In *Parallel and Distributed Computing, Applications and Technologies, 2007. PDCAT'07. Eighth International Conference on*, pages 179–180. IEEE, 2007.

[32] J.H. Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of the 1993 conference of the Centre for Advanced Studies on Collaborative research: software engineering-Volume 1*, pages 171–183. IBM Press, 1993.

[33] J.H. Johnson. Visualizing textual redundancy in legacy source. In *Proceedings of the 1994 conference of the Centre for Advanced Studies on Collaborative research*, page 32. IBM Press, 1994.

[34] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. *Software Engineering, IEEE Transactions on*, 28(7):654–670, 2002.

[35] M.E. Karim, A. Walenstein, A. Lakhotia, and L. Parida. Malware phylogeny generation using permutations of code. *Journal in Computer Virology*, 1(1):13–23, 2005.

[36] R.M. Karp. Combinatorics, complexity, and randomness. *Communications of the ACM*, 29(2):98–109, 1986.

[37] I. Keivanloo, C. K. Roy, J. Rilling, and P. Charland. Shuffling and randomization for scalable source code clone detection. In *Software Clones (IWSC), 2012 6th International Workshop on*, pages 82–83. IEEE, 2012.

[38] H. Kim, Y. Jung, S. Kim, and K. Yi. Mecc: memory comparison-based clone detector. In *Software Engineering (ICSE), 2011 33rd International Conference on*, pages 301–310. IEEE, 2011.

[39] R. Komondoor and S. Horwitz. Using slicing to identify duplication in source code. *Static Analysis*, pages 40–56, 2001.

[40] K.A. Kontogiannis, R. DeMori, E. Merlo, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Automated Software Engineering*, 3(1):77–108, 1996.

[41] J. Krinke. Identifying similar code with program dependence graphs. In *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*, pages 301–309. IEEE, 2001.

[42] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Recent Advances in Intrusion Detection*, pages 207–226. Springer, 2006.

[43] V.I. Levenshtein. Binary codes capable of correcting deletions, insertions. Technical report, and reversals. Technical Report 8, 1966.

[44] C. Liu, C. Chen, J. Han, and P.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. ACM, 2006.

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

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

[47] J. Mayrand, C. Leblanc, and E.M. Merlo. Experiment on the automatic detection of function clones in a software system using metrics. In *Software Maintenance 1996, Proceedings., International Conference on*, pages 244–253. IEEE, 1996.

[48] C.K. Roy and J.R. Cordy. A survey on software clone detection research. *Queens School of Computing TR*, 541:115, 2007.

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

[50] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su. Detecting code clones in binary executables. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 117–128. ACM, 2009.

[51] M. Schordan and D. Quinlan. A source-to-source architecture for user-defined optimizations. *Modular Programming Languages*, pages 214–223, 2003.

[52] A. Schulman. Finding binary clones with opstrings & function digests: Part 1-reverse engineering is an invaluable engineering tool. *Dr Dobb's Journal-Software Tools for the Professional Programmer*, pages 69–73, 2005.

[53] A. Schulman. Finding binary clones with opstrings & function digests: Part ii. *Dr. Dobb's Journal*, 30(8):56, 2005.

[54] A. Schulman. Finding binary clones with opstrings function digests: Part iii. *Dr. Dobb's Journal*, 30(9):64, 2005.

[55] D.M. Shawky and A.F. Ali. An approach for assessing similarity metrics used in metric-based clone detection techniques. In *Computer Science and Information Technology (ICCSIT), 2010 3rd IEEE International Conference on*, volume 1, pages 580–584. IEEE, 2010.

[56] R. Tairas and J. Gray. Phoenix-based clone detection using suffix trees. In *Proceedings of the 44th annual Southeast regional conference*, pages 679–684. ACM, 2006.

[57] C. J. van Rijsbergen. *Information Retrieval*. University of Glasgow, 1979.

[58] V. Wahler, D. Seipel, J. Wolff, and G. Fischer. Clone detection in source code by frequent itemset techniques. In *Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on*, pages 128–135. IEEE, 2004.

[59] A. Walenstein, M. Venable, M. Hayes, C. Thompson, and A. Lakhotia. Exploiting similarity between variants to defeat malware. In *Proc. BlackHat DC Conf*, 2007.

[60] Z. Wang, K. Pierce, and S. McFarling. Bmat: a binary matching tool for stale profile propagation. *The Journal of Instruction-Level Parallelism*, 2:1–20, 2000.

[61] R.M. Zeidman. Patent no. 2008/0270991a1. us., 2008.

[62] R.M. Zeidman. Patent no. 7823127b2. us., 2010.

[63] J. Zobel and A. Moffat. Exploring the similarity space. In *ACM SIGIR Forum*, volume 32, pages 18–34. ACM, 1998.