Contributors: Mohammad Reza Farhadi[a, c], Benjamin C.M. Fung[a], Yin Bun Fung[c], Philippe Charland[b], Stere Preda[c, 1], Mourad Debbabi[c]

a School of Information Studies, McGill University, Montreal, QC, Canada
b Mission Critical Cyber Security Section, DRDC-Valcartier Research Centre, Quebec, QC, Canada
c Concordia Institute for Information Systems Engineering, Concordia University, Montreal, QC, Canada

Title: Scalable code clone search for malware analysis

Published in: *Digital Investigation (DIIN): Special Issue on Big Data and Intelligent Data Analysis*

Citation: M. R. Farhadi and al. Scalable code clone search for malware analysis, Digital Investigation, Volume 15, December 2015, Pages 46–60, Special Issue: Big Data and Intelligent Data Analysis, doi:10.1016/j.diin.2015.06.001

Doi: doi:10.1016/j.diin.2015.06.001
http://www.sciencedirect.com/science/article/pii/S1742287615000705

Scalable Code Clone Search for Malware Analysis

Mohammad Reza Farhadi[a,c], Benjamin C. M. Fung[a], Yin Bun Fung[c], Philippe Charland[b], Stere Preda[c], Mourad Debbabi[c]

[a] *School of Information Studies, McGill University, Montreal, QC, Canada*
Email: {mohammad.farhadi, ben.fung}@mcgill.ca
[b] *Mission Critical Cyber Security Section, DRDC-Valcartier Research Centre, Quebec, QC, Canada*
Email: philippe.charland@drdc-rddc.gc.ca
[c] *Concordia Institute for Information Systems Engineering, Concordia University, Montreal, QC, Canada*
Email: {yb_fung, s_preda, debbabi}@ciise.concordia.ca

Abstract

Reverse engineering is the primary step to analyze a piece of malware. After having disassembled a malware binary, a reverse engineer needs to spend extensive effort analyzing the resulting assembly code, and then documenting it through comments in the assembly code for future references. In this paper, we have developed an assembly code clone search system called *ScalClone* based on our previous work on assembly code clone detection systems [1][2]. The objective of the system is to identify the code clones of a target malware from a collection of previously analyzed malware binaries. Our new contributions are summarized as follows: First, we introduce two assembly code clone search methods for malware analysis with a high recall rate. Second, our methods allow malware analysts to discover both exact and inexact clones at different token normalization levels. Third, we present a scalable system with a database model to support large-scale assembly code search. Finally, experimental results on real-life malware binaries suggest that our proposed methods can effectively identify assembly code clones with the consideration of different scenarios of code mutations.

Keywords: assembly code clone detection, malware analysis, reverse engineering

1. Introduction

Malware, such as viruses, trojan horses, and worms, is software that intends to gain unauthorized access to computer systems or information. As malware attacks become more frequent, there is a pressing need to develop systematic techniques to understand their behaviour and potential impacts. Although reverse engineering is often the primary step taken to gain an in-depth understanding of a piece of malware, it is a time-consuming and manual process. To achieve a more efficient analysis, malware analysts can manually compare the assembly code under study 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 previously analyzed to the new assembly code, analysts can minimize redundant efforts and focus their attention on the new functionalities of the malware. However, the comparison process itself is a challenging task, and the chances of identifying similar code fragments often depend on the experience and knowledge of the analyst.

The problem of assembly code clone search for malware is informally described as follows: Given a large repository of previously analyzed malware and a new target malware, the goal is to identify all code fragments in the repository that are syntactically or semantically similar to the code fragments in the target malware. This research problem was identified from extensive discussions with malware analysts and reverse engineers from the software industry and a government agency. The challenges and special requirements of clone search for malware analysis are summarized as follows.

Simple keywords matching insufficiency: A simple method to identify assembly code clones is to identify some keywords such as constants, strings, and imported function names in a code fragment, and then attempt to match them in the code repository. This method is applicable only if the target code fragments contain some uniquely identifiable numbers or strings.

Large code repository: As part of the reverse engineering process, malware analysts put comments in assembly files and archive them to a code repository. The size of an assembly code repository can range from a couple of megabytes to over dozens of gigabytes of textual data, and the size keeps growing as new malware emerges. A clone search system for malware analysis should be efficient and scalable. Efficiency refers to the response time of identifying all the clones of a target file from a code repository. Scalability refers to its capability to handle the size of a code repository.

Robustness: Malware with obfuscated code and different compiler optimizations may degrade the clone search accuracy. The optimization options used to compile the target malware binary in hand and the optimization options used to compile the malware binaries in the repository may be different. Thus, a clone search system for malware analysis should be robust enough to handle these variations.

Consistency: To support effective malware analysis, the clone search results should be consistent and repeatable, given the same search query and malware repository. In other words, the clone search methods should be deterministic. Otherwise, it would be very difficult for an analyst to draw meaningful conclusions and preserve the evidence from the results.

The literature [3] on source code clone detection defines four types of clones. Types I, II, and III are known as textual clones, while Type IV clones are known as semantic clones. We borrow a similar notion from source code clone detection and define the following four types of clones for assembly code clone detection.

Type I: Identical code fragments except for variations in whitespace, layouts, and comments.

Type II: Structurally and syntactically identical fragments except for variations in memory references, registers, constant values, whitespace, layouts, and comments.

Type III: Copied fragments with further modifications. Statements can be changed, added, or removed, in addition to variations in memory references, registers, constant values, whitespace, layouts, and comments.

Type IV: Code fragments that perform the same computation, but implemented using different syntactic variants.

Contributions: In this paper, we have *significantly* improved our previous assembly code clone detection system *BinClone* [1][2], which was developed based on the framework suggested by Sæbjørnsen et al. [4]. *BinClone* provides a proof of concept on the feasibility of detecting both exact and inexact clones in assembly code, but it was not designed for handling large volumes of assembly code. The proposed clone search system, *ScalClone*, addresses the aforementioned challenges for malware analysis. To the best of our knowledge, this is the first scalable clone search system specifically designed for supporting malware analysis. The contributions of *ScaleClone* are summarized as follows:

- *High recall rate.* Sæbjørnsen et al. [4] presented a method to identify

Type III clones, i.e., clone pairs that are not exactly identical. Their general approach is to first extract a set of features from each region, create a feature vector for each of them, and then use *locality-sensitive hashing* (*LSH*) to find the nearest neighbor vectors of a given query vector. Although their approach shows some encouraging results in identifying Type III clones, its assumption on the uniform distribution of vectors may not hold as the number of features (i.e., dimensions) increases, resulting in many false negatives. In contrast, our proposed inexact clone search method addresses the issue of false negatives and improves the recall rate by employing a filtering process to identify Types I, II, and III clones.

- *Scalable clone search system.* The clone search process consists of two phases. The first phase is to extract features from the previously analyzed assembly code and populate the feature values into a database. The second phase is to search for clone fragments of a target assembly file from the database. The computational complexity of both phases is linear to the size of the assembly code repository. Experiments on a large collection of real-life binary (assembly) files suggest that our proposed system is scalable.

- *Support for malware analysis.* Malware analysts clearly stated two requirements. First, the clone results have to be deterministic. Thus, any clone search method that employs randomization, such as the nondeterminisitic LSH in [4], does not satisfy this requirement. In this paper, we present a deterministic inexact clone detection method while maintaining its scalability. Second, the system should support clone search at different normalization levels of assembly code tokens. Our proposed normalization process allows the user to generalize the tokens, i.e., memory references, registers, and constant values, to different levels. Experimental results suggest that generalizing tokens can help improve the recall rate when identifying Types I and II clones.

- *Robustness against obfuscation techniques and different compiler optimizations.* Malware authors often want to hide the true intention of their code and sometimes employ obfuscation techniques, such as *instruction reordering* and *do-nothing code insertion*, to disorient the attention of malware analysts. Furthermore, the same piece of source code compiled using different compiler optimization options results in

different assembly code. Experimental results suggest that our proposed clone search technique is robust against obfuscated code clones and different compiler optimizations.

- *Systematic evaluation of assembly code clones.* Measuring the accuracy of the assembly code clone results is challenging because it is infeasible to manually identify all assembly code clones as ground truth. Thus, we propose an approach to systematically evaluate the accuracy of assembly code clones with respect to source code clones. The general idea is to insert a unique identifier to each block in the source code, apply a source code detection method to identify all source code clones, and then evaluate the quality of the assembly code clones with respect to the source code clones.

The rest of this paper is organized as follows: Section 2 describes related work and some background information on clone detection, followed by a formal problem definition of assembly code clone search in Section 3. The proposed clone search system is described in Section 4. Section 5 provides more details on the database schema. Section 6 presents the experimental results of the implemented system. Section 7 concludes the paper.

2. Related Work

The problem of source code clone detection has been studied extensively in the literature. The methods can be broadly categorized into seven categories, namely *token-based* [5][6], *text-based* [7][8], *similarity distance-based* [9], *metric-based* [10][11], *tree-based* [12][13], *program dependency graph (PDG)-based* [14][15] and *hybrid* [16][17] approaches. However, the existing source code clone detection techniques are inapplicable to assembly code because source code contains different types of control flow statements, such as while-loop, for-loop, switch-case, and if-then-else.

In contrast, assembly code contains a large collection of instructions that share a nearly identical format. The assembly code clone detection methods can be broadly categorized as follows:

Text-based approach: Jang et al. [18][19] used a fingerprinting algorithm based on bloom filters to cluster malware samples. While their work does not directly concern the comparison of unknown assembly code to previously analyzed samples, it does present a potential solution to address the scalability of any comparison system. Rahimian et al. [20] proposed a system called

RESource to match a piece of assembly code with online source code repositories, in order to identify the libraries used in the target binary file. The general idea is to extract keywords from the target assembly code and use them to query open source code repositories. *RESource* cannot find clones in assembly code as proposed in this paper.

Token-based approach: Schulman [21] proposed a system to create a database of previously analyzed binaries to recognize duplicate functions. This is a pioneer work on detecting code clones at the functional level. Karim et al. [22] addressed the problem of classifying new malware into existing malware families whose individual entries share common code. Walenstein et al. [23] further extended this approach to match assembly code by comparing *n-grams* and *n-perms* extracted from disassembled binaries that have been unpacked and decrypted. Each binary file is represented as a feature vector for similarity comparison. The method is effective only if two binary files are similar. In contrast, our proposed method works at the fragment level.

Metric-based approach: Bruschi et al. [24] presented a technique to normalize assembly code and detect both polymorphic and metamorphic malware. The prototype is based on using a normalization technique to ease the comparison between malware samples. The prototype implementation has some restrictions regarding the identification of program characteristics such as removing obfuscated instructions and dead code. These restrictions can have a huge impact on the results. Sæbjørnsen et al. [4] presented a general clone detection framework that utilizes an existing tree similarity framework, models the assembly instruction sequences as vectors, and groups similar vectors together using existing "nearest neighbor" algorithms.

Structural-based approach: Dullien et al. [25] presented research results on executable code comparison for attacker correlation. They implemented a system that can identify code similarities in executable files. Carrera and Erdelyi [26] 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. Briones et al. [27] 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. Flake [28] presented a method that constructs an isomorphism between the groups of functions that are used into two different versions of the same binary. Dullien

developed *BinDiff*[1] which is a software tool that compares binary files using a graph-based approach and displays the matched functions in a control flow graph. It parses and extracts features of every function, and then iteratively matches the callers and callees of a function. *BinDiff* assumes the input binaries are different versions of the same piece of code. The main purpose of *BinDiff* is to analyze software patches. *BinCrowd*[2] creates a repository that stores the analyzed assembly code together with their function names, comments, or other related information. This tool employs the *BinDiff* algorithm to discover known functions in other disassembled files.

Behavioral-based approach: Comparetti et al. [29] developed a system called *REANIMATOR* that allows the identification of dormant functionalities in malware. The system exploits the fact that a dynamic malware analysis captures malware execution and reports its behavior. This approach can be useful if one wants to match different code portions that are semantically identical but syntactically different. Jin et al. [30] proposed a binary function clustering method to group similar functions with respect to their behaviour. Their method first captures the semantic of each function using the machine state changes made at the functional level. Then, a clustering method using hashing is used to identify code clones based on common features. Bayer et al. [31] proposed a method to group malware by their behaviour, but this problem is different from assembly clone detection.

Hybrid approach: Wang et al. [32] presented a tool called *BMAT* that creates mappings between old and new versions of binaries. This tool is used to generate the profile information of applications and illustrates how to propagate stale profiles from an old to a new version. Khoo et al. [33] developed a search engine that allows searching for binary code against some existing open source code repositories. First, they tokenized assembly functions and then extracted three sets of features to build a query term. These features include mnemonics, CFGs, and constant values. As they considered only assembly functions as the unit of comparison, their work cannot find code clones within functions. Their experiments were not performed on malware, as we do in this paper, but their results on GNU C libraries show that their search engine can efficiently identify the matched binary code with a high recall rate, with the trade-off of relatively low precision. Their search engine

[1] http://www.zynamics.com/bindiff.html
[2] http://www.zynamics.com/bincrowd.html

does not employ normalization for assembly operands. Therefore, it may not be able to correctly identify Types I and II clones.

3. Problem Definition

The problem of assembly code clone search is formally defined as follows: Let $A = \{A_1, \ldots, A_n\}$ be a collection of analyzed and stored assembly files, where each assembly file A_i consists of a collection of functions F and each function $f \in F$ contains one or multiple lines of assembly code instructions. 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 inclusively in function f in A_i. $|f[a : b]|$ denotes the number of lines of assembly code in $f[a : b]$. In the rest of this section, we assume the memory references, registers, and constant values in the assembly code have been normalized. The normalization process is explained in Section 4.

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

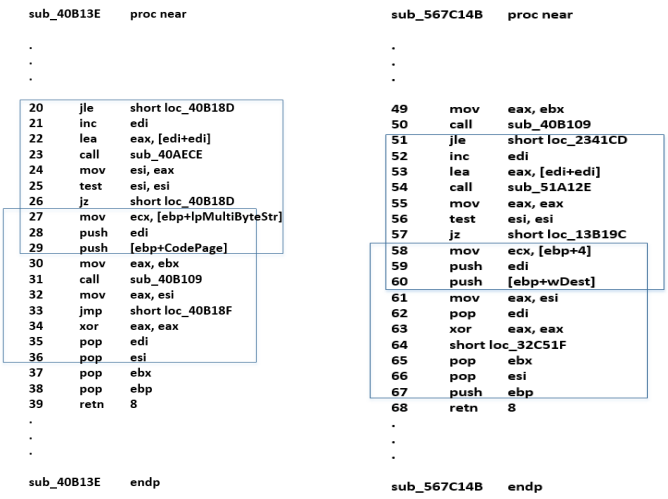
Definition 3.1 (Exact clone). *Let $f1[a : b]$ and $f2[c : d]$ be two arbitrary non-empty code fragments in functions $f1$ and $f2$, respectively. $f1[a : b]$ and $f2[c : d]$ are an* exact clone pair *if $|f1[a : b]| = |f2[c : d]|$ and $f1[a] = f2[c], \ldots, f1[b] = f2[d]$. The relation $=$ denotes that two code fragments are identical with respect to the sequence of mnemonics and normalized operands appearing on the assembly code instruction line.* ■

Definition 3.2 (Inexact clone). *Let $f1[a : b]$ and $f2[c : d]$ be two arbitrary non-empty code fragments in functions $f1$ and $f2$, respectively. Let $sim(f1[a : b], f2[c : d])$ be a function that measures the similarity between two code fragments $f1[a : b]$ and $f2[c : d]$. $f1[a : b]$ and $f2[c : d]$ are an* inexact clone pair *if $sim(f1[a : b], f2[c : d]) \geq minS$, where $minS$ is a user-specified minimum similarity threshold $0 \leq minS \leq 1$.* ■

The problem of assembly code clone search is formally defined as follows.

Definition 3.3 (The problem of assembly code clone search). *Let $A = \{A_1, \ldots, A_n\}$ be a collection of previously analyzed assembly files. Let tar be*

a target assembly file or a target assembly code fragment. The problem of assembly code clone search is to identify all exact and inexact clone pairs of tar from A, given a minimum similarity threshold minS. ■



(a) sub_40B13E (b) sub_567C14B

Figure 1: Examples of exact clone and inexact clone

Example 1. *Figure 1 shows two assembly code fragments extracted from* Zeus*, which is a well-known trojan horse that uses the man-in-the-browser keystroke logging and form grabbing techniques to extract banking information. Suppose* sub_40B13E *is the target, and* sub_567C14B *is in the repository. Code fragments* $f[20, 29]$ *and* $f[51, 60]$ *are an exact clone pair. Code fragments* $f[27, 36]$ *and* $f[58, 67]$ *are an inexact clone pair.* ■

Note that an exact clone pair has $sim(f1[a : b], f2[c : d]) = 1$. In other words, an exact clone pair is also an inexact clone pair with similarity equal to 1. 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 search methods for identifying exact and inexact clones separately. This observation is incorrect because an inexact clone pair with $sim = 1$ does not necessarily mean they are an exact clone pair. Consider two normalized code fragments that contain two identical instructions but in different order. They have $sim = 1$, but they are not an exact clone pair. In real-life malware analysis,

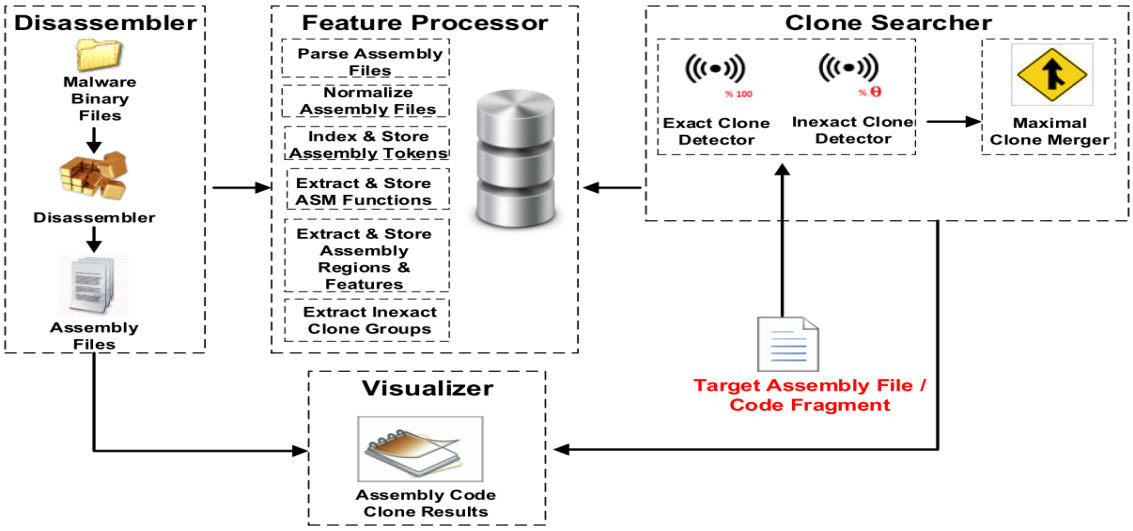


Figure 2: Overview of *ScalClone*

an analyst might want to identify only exact clones. Thus, the two clone search methods should remain separate.

4. Assembly Code Clone Search System: *ScalClone*

Figure 2 depicts an overview of the proposed assembly code clone search system *ScaleClone*, which consists of four major components, namely *disassembler*, *feature processor*, *clone searcher*, and *visualizer*. The first step is to disassemble a collection of malware binaries into a collection of assembly files. The feature processor extracts features from the assembly files and loads them into a database. The clone searcher takes as input a target assembly file or a target code fragment and identifies its clone pairs from the database. Finally, the clone visualizer displays the identified clone pairs and allows a user to interactively browse through them.

4.1. Disassembler

This step disassembles the input binaries into a collection of assembly files A using a disassembler such as IDA Pro[3]. Each assembly file $A_i \in A$ contains a set of functions. Each function contains a sequence of assembly

[3]http://www.hex-rays.com/products/ida

code 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: *memory reference*, *register reference*, and *constant value*.

4.2. Feature Processor

The objective of the feature processor is to extract features from a collection of assembly files A and load them into a database in order to support code clone search. Specifically, the feature processor performs six steps on each assembly file in A: (i) Each assembly file is parsed and irrelevant assembly code and comments are removed. (ii) The assembly code is normalized for clone comparison. (iii) *Constants*, *strings*, and *imports* tokens in the assembly code are indexed to support token search. (iv) Functions (or subroutines) from the assembly are extracted, based on the function header and footer, as indicated by the keywords *proc near*, *proc far*, and *endp*. (v) Each function is partitioned into an array of *regions* and features from each region are extracted for inexact clone detection. (vi) The regional features vectors are stored into the database. The rest of this section provides more details on some of these steps.

4.2.1. Normalize Assembly Files

Referring to the description of Type II clones in Section 1, two structurally and syntactically identical fragments with variations in *memory references*, *registers*, and *constant values* can be considered a clone pair.

Two code fragments may be considered an exact clone pair even if some of their operands are different. For example, two instructions with the same mnemonic but with different registers, such as `eax` and `ebx`, can be considered identical. 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 selected by the user. For constant values, the normalizer generalizes them to VAL, which simply ignores the exact constant value. The same logic applies to memory references. For registers, the user can generalize them according to the normalization hierarchy depicted in Figure 3. The top-most level *REG* generalizes all registers, regardless of 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 three groups by size: 32-, 16-, and 8-bit registers.

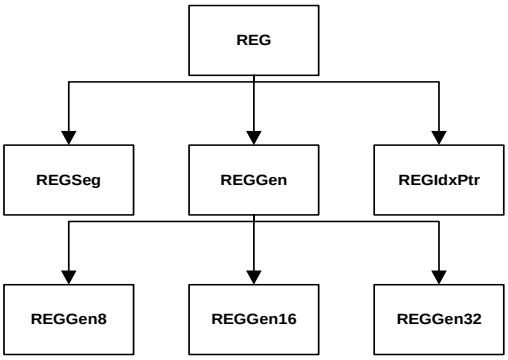


Figure 3: Normalization hierarchy for registers

Example 2. *Figure 4 shows the normalized version of the sub_40B13E function in Figure 1 based on the* REG *normalization level.* ■

```
sub_40B13E      proc near

.
.
.

20      jle     short loc_40B18D
21      inc     REG
22      lea     REG, MEM
23      call    sub_40AECE
24      mov     REG, REG
25      test    REG, REG
26      jz      short loc_40B18D
27      mov     REG, MEM
28      push    REG
29      push    MEM
30      mov     REG, REG
31      call    sub_40B109
32      mov     REG, REG
33      jmp     short loc_40B18F
34      xor     REG, REG
35      pop     REG
36      pop     REG
37      pop     REG
38      pop     REG
39      retn    VAL

sub_40B13E      endp
```

Figure 4: Normalized assembly code sample used in Zeus

4.2.2. Extract Assembly Regions and Features

Each function is partitioned into an array of overlapping regions using a sliding window with a size of at most w instructions, where w is a user-specified threshold. Figure 5 shows the extracted regions of the normalized procedure sub_40B13E in Figure 4 with $w = 15$. The offset between regions is set to 1 by default in order to ensure that no regions are skipped. Therefore, no clones will be missed. Each region is then hashed to an unsigned integer and stored into the database *Region* table. Section 5 discusses the database schema in details.

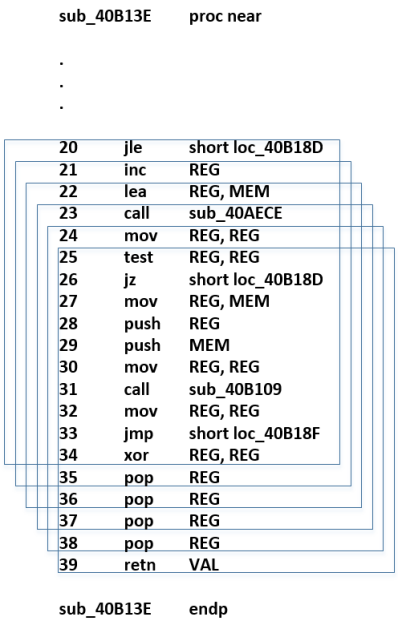


Figure 5: Regionization for $w = 15$

4.2.3. Extract Region Features

This step consists of three operations. (i) Construct a feature frequency vector for each region. A feature frequency vector contains the frequencies of three groups of features from the assembly instructions [4]. The first group of features includes all mnemonics. The second group includes all operand types. The third group includes all combinations of mnemonics and the type of the first operand. (ii) Compute the median of each feature over all regions. The median feature vector is stored into the database *Feature* table.

The medians serve as the separation points for grouping regions because the frequency distribution of the features is skewed and the median is a good measure for central tendency of skewed data. If more than half of the regions do not contain a particular feature, then the median of the feature is zero and the feature cannot be used to differentiate regions. Thus, all features that have medians equal to zero are removed from the median vectors. Experiments suggest that this will significantly reduce the size of the vectors in the subsequent steps. (iii) Store the median feature vector into database table *FilteredFeature*.

4.2.4. Store Inexact Clone Groups

Intuitively, two assembly code fragments are considered as an inexact clone pair if they share many similar, but not exactly the same features in terms of mnemonics and operand types. The objective of this step is to group regions into overlapping clone groups by their syntactic similarity. Let *med* be the non-zero median vector. For each region, construct a binary vector by comparing its feature frequency vector with the non-zero median vector *med*. All features with a zero median are skipped. If the feature's value is larger than the corresponding median in *med*, then 1 is inserted into the corresponding entry of the binary vector. Otherwise, 0 is inserted. To build the clone groups, we compute all possible two-combinations of the binary vectors. Let s be the size of the binary vectors. There are $C(s,2) = \frac{s \times (s-1)}{2}$ possible two-combinations. Each combination has the size of 2 with $2^2 = 4$ possible values, namely $(0,0)$, $(0,1)$, $(1,0)$, and $(1,1)$. Each of these combinations forms a *clone group*, making $C(s,2)$ number of clone groups. Each clone group maps each region to one of the 4 possible values. These clone groups will be used for inexact clone detection in Section 4.3.2. These clone groups are then inserted into the database table *InexactGroups*.

Example 3. *Figures 6 and 7 show a feature vector, a median vector, and a binary vector of size 5 for a region. As the size of the median vector is 5, there are* $C(5,2) = 10$ *clone groups. Figure 7 shows the 10 clone groups with the values the sample region is mapped to.* ■

4.3. Clone Searcher

This component aims to search for possible clones of a target assembly file or code fragment. First, we normalize the target file or target code fragment, extract its functions, and regionize them based on the same algorithms

Feature Vector:	mov	call	push	movREG	callMEM
Median Vector:	8	5	1	1	4
Binary Vector:	1	0	1	1	0

Figure 6: Sample feature, median and binary vector of size 5

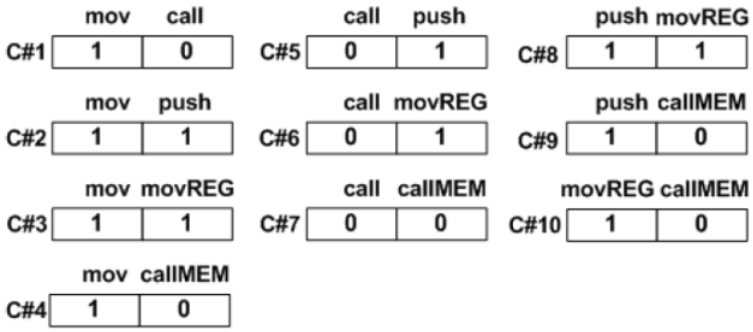


Figure 7: Sample clone groups

explained in Section 4.2. Then, the clone search sends each extracted region to both exact and inexact clone detectors to search for possible clones. The last step is to merge and unify the clones found.

4.3.1. Exact Clone Detector

Refer to Definitions 3.1 and 3.2. A clone pair is defined as an unordered pair of code regions that have similar normalized statements. This step identifies the exact clone pairs of the target regions. Two regions are considered an exact clone pair if all the normalized statements in the two regions are identical. We used a hashing approach with linear complexity to find the exact clones of a target region. Specifically, if a target region shares the same hash value with a region, they are considered to be an exact clone pair. As this approach uses a hash algorithm to map each region to an integer value, all identical regions are mapped to the same hash value without false negatives. The process requires only one scan of the regions. A polynomial hash function, called *djb2* is used for this purpose. *djb2* is defined as follows:

$$h(0) = 5381,\ h(k) = 33h(k) + s(k)\ for\ k > 0 \tag{1}$$

where s_k denotes the k$^{\text{th}}$ character of the string being hashed [34]. It simply starts with a seed, multiplies it by 33 and adds the current character to

create a new hash value. Then it repeats this process for every character in the string.

Algorithm 1 provides the details of the method. First, the target regions are hashed to unsigned integers v based on their instructions. Then, regions with their corresponding hash values are stored in the database. In Lines 3-5, the method iterates through each target region r, creates a hash value v, and adds the region r together with its hash value to a hash table H_{target}. In Lines 6-9, the method iterates through each bucket in (H_{target}), finds the similar regions in terms of hash value with the regions stored in D_{exact}, and forms an array of exact clone pairs, denoted by EP.

Algorithm 1: Exact Clone Detector

```
   input  : target assembly file A
            database table D_exact
   output: set of exact clone pairs EP
 1 begin
 2     H_target ← ∅;
 3     foreach region r ∈ A do
 4         v ← hash(r);
 5         H_target ← {r, v};
 6     foreach record d ∈ D_exact do
 7         foreach h ∈ H_target do
 8             if h.v = d.hashValue then
 9                 EP ← EP ∪ {(h.r, d.region)};
10     return EP;
```

4.3.2. Inexact Clone Detector

The objective of the inexact clone detector is to identify inexact clone pairs of a target assembly file/fragment from a given collection of regions. For each region in each function of the target assembly file or code fragment, a binary vector is constructed as described in Section 4.2.4. Then, for every combination of two features in the binary vector, the inexact clone detector submits a query to the database to retrieve all regions from the *InexactGroups* table that match the two features and binary values of the target region. Each two combination forms a clone group. The database query Q is:

```
SELECT dbRegionID FROM InexactGroups
WHERE featAIDFKey = firstFeatureID AND
      featBIDFKey = secondFeatureID AND
      featAPresent = 1/0 AND
      featBPresent = 1/0
```

where *featAIDFKey* is the identifier of the first non-zero median feature; *featBIDFKey* is the identifier of the second non-zero median feature in the binary vector; and *featAPresent* and *featBPresent* represent the presence or absence of the features in the target region.

A co-occurrence counter is created for each stored region to keep track of the number of co-occurrences of the stored region with the target region. In other words, the co-occurrence counter keeps track of the number of common inexact clone group values between the stored region and the target region. The stored regions with the number of co-occurrences equal to or greater than the minimum similarity threshold $minS$ times the total number of clone groups, i.e., $minS \times C(n, 2)$, are considered to be the inexact clone pairs of a target region, where n is the size of a binary vector.

Algorithm 2 provides an overview of the inexact clone detection method in four steps.

- **Step #1: Extract target file features**: This step extracts the features of each region in the target file. It follows the same method explained in Section 4.2.3 to construct a feature vector.

- **Step #2: Generate target file binary vectors**: This step constructs a binary vector for each target region by comparing the feature vector of the target region with the median attribute of the *FilteredFeature* table. 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.

- **Step #3: Count co-occurrences**: For each target region r, this step first retrieves the clone groups of r, denoted by CG, by executing a database query Q on the $InexactGroups$ table. A region r' *co-occurs* with a target region r if they share the same value in the retrieved clone group. This step counts the number of co-occurrences.

- **Step #4: Identify inexact clone pairs**: The regions having the number of co-occurrences above or equal to the similarity threshold

Algorithm 2: Inexact Clone Detector

input : target assembly file A
minimum similarity threshold $minS$
database table $D_{FilteredFeatures}$
database table $D_{InexactGroups}$
output: set of inexact clone pairs IP

```
1  begin
2      // Step1
3      foreach region r ∈ A do
4          r.F ← extractFeatures(r);
5      // Step2
6      foreach record d ∈ D_FilteredFeature do
7          foreach region r ∈ A do
8              if r.F[d] ≥ d.Median then
9                  r.binary[d] ← 1;
10             else
11                 r.binary[d] ← 0;
12     k = C(n, 2);
13     foreach region r ∈ A do
14         // Step3
15         CG ← D_InexactGroups.Q(r);
16         foreach clone group cg ∈ CG do
17             foreach region r' ∈ cg do
18                 ++r'.cooccurCnt;
19                 // Step4
20                 if r'.cooccurCnt ≥ minS × k then
21                     IP ← IP ∪ {(r, r')};
22     return IP;
```

$minS \times C(n, 2)$ are considered to be the inexact clone pairs, where n is the size of the binary vector.

4.3.3. Maximal Clone Merger

Since the clone search process operates on regions, the size of the identified clones is bounded by the window size w. As a result, a naturally large clone fragment may be broken down into small consecutive clone regions, making the malware analysis difficult. The objective of this step is to merge the smaller consecutive clone regions into a larger clone. Algorithm 3 presents the maximal clone merger process. It takes a set of clone pairs CP as input and returns a set of maximal merged clone pairs MP as output. The clone pairs in CP are sorted by line numbers of the target file/fragment in ascending order. The function *overlap* determines whether or not two regions have an overlap. Two consecutive clone pairs are merged into a single clone pair if their regions in the target file *and* in the stored file have an overlap. The *overlap* function in this algorithm works differently for exact and inexact clone detection. If the identified exact clones have an overlap, they are merged. If the identified inexact clones have an overlap, their overall size after the merge may differ. This may happen if the offset between each overlapped clone is not the same. In this case, two inexact clone pairs are merged if the difference between the offsets of two regions is smaller than or equal to $(1 - minS) \times windowSize$.

Algorithm 3: Maximal Clone Merger

input : sorted sequence of clone pairs CP
output: set of maximal merged clone pairs MP

```
1 begin
2     MP ← CP;
3     for every two consecutive clone pairs cp, cp' ∈ MP do
4         if overlap(cp.A, cp'.A) and overlap(cp.B, cp'.B) then
5             MP ← merge(cp, cp');
6     return MP;
```

Example 4. *Figure 8 provides an example of the maximal clone merger process. 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 six clone pairs. Since every consecutive clone pair overlaps, they are merged into one clone pair, as indicated by the outer dashed rectangles.* ■

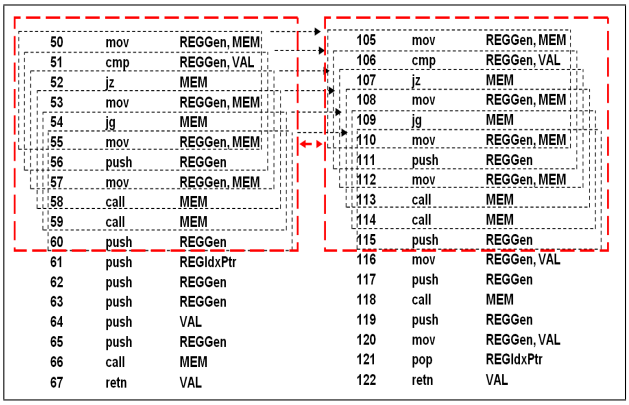


Figure 8: Maximal clone merging with $w = 5$

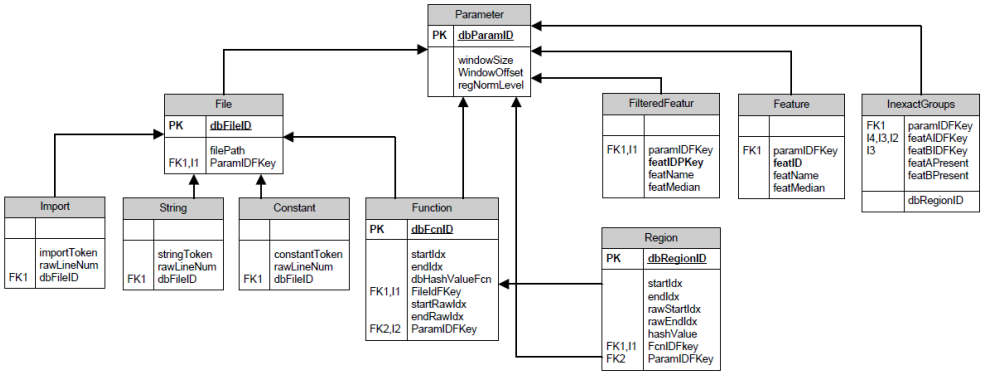


Figure 9: Database model for clone search system

5. Database Model for Clone Search System

Figure 9 depicts the relational database model of our implementation. Our proposed system allows malware analysts to employ different parameters: window size, window offset, and register normalization level. Every combination of these parameters forms a record in the *Parameter* table. Each assembly file contains a set of functions. Each function is represented as a set of overlapping regions. Their corresponding attributes are stored into the *File*, *Function*, and *Region* tables, respectively. The *Feature* table stores the extracted features together with their median. The *FilteredFeature* table stores the non-zero median features discussed in Section 4.2.3. Finally, the *InexactGroups* table contains the regions and their clone groups generated in Section 4.2.4.

In the preprocessing step, the preprocessor may remove some code instructions and comments from the assembly files. Yet, when the visualizer displays a clone pair, malware analysts prefer viewing the original assembly files. Thus, we need to keep track of the line numbers before and after the processing step in order to correctly display the code in the visualization step. Malware analysts are particularly interested in searching for imports, strings, and constants. Thus, a separate index is built for them to provide efficient retrieval. They are stored in the *Import*, *String*, and *Constant* tables, respectively.

6. Experimental Evaluation

The objective of the experimental study is to evaluate the proposed assembly code clone search system, *ScalClone*, in terms of accuracy, efficiency, and scalability. Extensive experiments were conducted on four data sets, namely *DLL18*, *zlib*, *malware297*, and *DLL1GB* in order to evaluate the performance of *ScalClone* with respect to different parameter values, obfuscation techniques, and compiler optimization options. We also compare our results with a previously proposed assembly code clone detection method [4], which uses *locality-sensitive hashing* (*LSH*) for identifying inexact clones. All experiments were performed on a PC with an Intel Xeon E31220 3.10GHz Quad-Core processor, 16GB of RAM, running Microsoft Windows 7 64-bit.

6.1. Accuracy

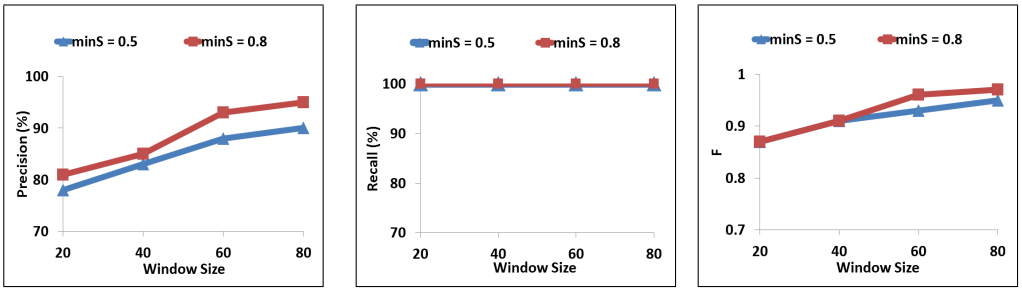
We employ three typical measures to evaluate the accuracy of the proposed clone search system:

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

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

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

where *Solution* is the set of a priori known clone fragments, *Result* is the set of code fragments in a clone search 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*.



(a) Precision (b) Recall (c) F-measure

Figure 10: Accuracy for *DLL18*

6.1.1. Data set: DLL18

The first data set, denoted by *DLL18*, is an assortment of 18 DLL files obtained from an operating system. The size of their assembly code is not too large. Thus, we can afford to manually identify the a priori known clone fragments. To evaluate the accuracy of the proposed clone search system, some code fragments were first selected from the 18 assembly files. Then, clones of the code fragments were manually identified in the data set. Finally, the results of the proposed system were compared with the manually identified clones in order to compute the precision, recall, and F-measure.

Figure 10 shows the precision, recall, and F-measure for minimum similarity threshold $minS = 0.5$ and $minS = 0.8$ on *DLL18*. The precision and F-measure are consistently above 75%. The recall is 100% for both exact and inexact detections, suggesting that the clone detection methods are effective.

6.1.2. Data set: zlib

The second data set is *zlib* v1.2.7 and v1.2.8[4], an open source library for data compression. It is a crucial component of many software platforms including Linux, Mac OS X, and iOS. The zlib project includes 235 files with total of 32,885 lines of source code. The compiled zlib includes 50,848 lines of assembly code with 221 functions. Due to the size of zlib, it is infeasible to manually identify a priori all the code clones. Thus, we propose a process to automate the evaluation. First, we employ *CCFinder*[5], a token-based source code clone detection tool [35], to first identify the source code clones and then

[4]http://www.zlib.net
[5]http://www.ccfinder.net

evaluate whether or not our clone search system can identify in the assembly code the ones identified by *CCFinder*. In order to perform this evaluation, we need a mechanism to match source code clones with assembly code clones. The assembly code clones that are indirectly identified by *CCFinder* are considered to be the *Solution* in the calculation of *Precision*, *Recall*, and *F-measure*.

In brief, we insert a unique identifier (a constant integer) into each source code block of zlib, use *CMake*[6] to make the zlib source code, compile zlib in Microsoft Visual Studio 2010 (64-bit), and reidentify the clones based on the unique identifiers. Figure 11 shows an example of the identifier insertion process in the source code and its corresponding assembly code. If the assembly clones identified by *ScalClone* are covered by the *Solution* of *CCFinder*, the clones are considered to be correctly identified.

```
if (state->out == NULL) {
    int myNewConst777 = 0x777;
    free(state->in);
    gz_error(state, Z_MEM_ERROR, "out of memory");
    return -1;
}
```

```
jnz     short loc_18000EA64
mov     [rsp+68h+var_14], 777h
mov     rax, [rsp+68h+state]
mov     rcx, [rax+30h]
call    cs:__imp_free
lea     r8, aOutOfMemory_4
mov     edx, 0FFFFFFFCh
mov     rcx, [rsp+68h+state]
mov     eax, 0FFFFFFFFh
jmp     loc_18000EB70
```

Figure 11: Token insertion

Exact clones: *CCFinder* can identify 79 source code clones between zlib v1.2.7 and v1.2.8. We manually review them and confirm that 62 of them are valid source code clones. The others are too small, with few source code lines, or are simply false positives. All 62 clones are Type I or Type II exact clones. We populate the assembly code features of zlib v1.2.7 into the database, and consider v1.2.8 as target files. The validity of the found clones are verified by inspecting if the clones are covered by the *Solution*. The mnemonics sequences for all clones are the same but some registers are different because of different register allocation techniques used by the compiler. These differ-

[6]http://www.cmake.org

ences are removed by our normalization technique described in Section 4.2.1. In this particular experiment, all the clones identified by *CCFinder* can also be identified by *ScalClone*. All of them are exact clones. Thus, both recall and precision of *ScalClone* are 100%.

Inexact clones: The objective of this experiment is to evaluate the performance of the inexact clone search method. In malware analysis, finding code clones that are mutations of a known piece of malware is a major challenge. The mutations can be caused by code obfuscation, instruction reordering, or different compiler optimization options:

- Do-nothing code insertion. One of the obfuscation techniques commonly used in malware is to insert some useless or do-nothing instructions in the assembly code. While they do not effect the overall functionality of a basic block or function, they make the analysis more difficult. This technique may bypass anti-virus detection based on software signatures.

- Instruction reordering. There are many reasons for instruction reordering, such as compiler optimization, as a side effect of different platforms, or as code obfuscation from malware authors.

- Different compiler optimization. Employing different compiler optimization options may result in different assembly code, but without changing its functionality. The question is whether or not the assembly code clones can still be identified.

To evaluate the effects of the above-mentioned mutations on assembly code clone search, we populate zlib v1.2.7 into the database and create three variants of v1.2.8, one variant for each mutation process. For *do-nothing code insertion*, we use *PELock*[7] to insert some do-nothing code into each of the 62 code clone fragments. This insertion mechanism does not change the functionality of the assembly code. *PELock* is a tool to modify x86 assembly code to make its analysis more difficult. For the second mutation, we reorder instructions using *PELock* without changing the functionality of the code fragments. For the third mutation, we first compile zlib 1.2.8 with a full optimization level using Visual Studio and then disassemble it using IDA Pro. This optimization option replaces and reorders some instructions

[7]http://www.pelock.com

due to pipeline optimizations. Although the assembly code is different, the functionality remains the same. We select all of the 62 code clone fragments and transform them based on these mutations. Then, we run *ScalClone* on the code fragments and examine them to see if the system can still find their corresponding code clones.

Method	Do-nothing	Reordering	Optimization
ScalClone Inexact Algorithm	90%	100%	62%
Locality Sensitive Hashing (LSH)	67%	100%	14%

Table 1: Comparison with LSH on *zlib*

Table 1 presents the recall rates of *ScalClone* for the three mutation scenarios. The rates for *do-nothing code insertion*, *instruction reordering*, and *compiler optimization* are 90%, 100%, and 62%, respectively. The recall rate for *do-nothing code insertion* is high because adding more code to the regions only adds more syntactic features, but the original features do not change much. The second scenario reorders the instructions. The recall stays at 100% because reordering has no impact on the frequencies of the mnemonics. Therefore, the syntactic features remain intact in most cases. In contrast, changing the compiler optimization options does affect the syntactic features. As a result, the impact on recall is more significant than in the other two scenarios.

LSH Comparison: Sæbjørnsen et al. [4] presented an inexact clone detection method using *locality-sensitive hashing (LSH)* to find the nearest neighbor vectors of a given query vector. Their assumption on the uniform distribution of vectors in the 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 is mapped to 0. Otherwise, it is mapped 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 using the following equation:

$$lsh(v) = (h_1(v), h_2(v), \ldots, h_m(v)) \tag{5}$$

The LSH method splits a vector space into 2^m sub-spaces by m base vectors. These base vectors are randomly chosen, and the distribution of vectors is 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 the chosen parameters, which is challenging in high-dimensional feature vectors. Also, due to the use of randomization, the clone results produced by LSH are non-deterministic. Some malware analysts clearly indicate that this non-deterministic behaviour is unacceptable because it will be very difficult for a reverse engineer to produce a consistent malware analysis for evidence preservation. To avoid the non-deterministic behaviour as in LSH, our proposed method employs fixed parameters derived from the data. The first one is the number of subspaces, which is the number of two-combinations of features. The second parameter is the subspace dimensionality, which is 2. In our comparison, we used the LSH algorithm implemented by Andoni and Indyk [36] and compared its results with *ScalClone* results for all the three aforementioned mutations.

Table 1 presents the recall rates for LSH in the three mutation scenarios. The rates for *do-nothing code insertion*, *instruction reordering*, and *compiler optimization* are 67%, 100%, and 14%, respectively. This result suggests that *ScalClone* in general is more robust than LSH with respect to do-nothing code insertion and compiler optimization.

6.2. Efficiency and Scalability

Next, we evaluate the efficiency and scalability of our proposed clone search system and compare them with our previous assembly code clone detection method *BinClone* [1]:

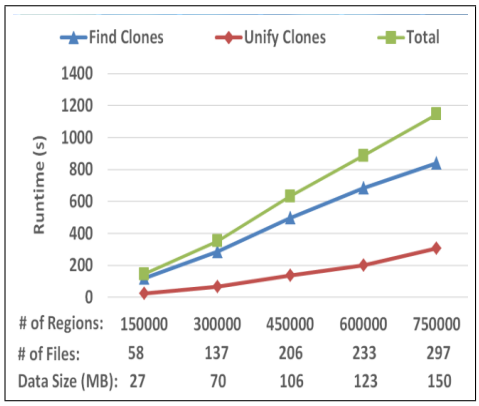
Efficiency and Scalability on *malware297*: The third data set is an assortment of 297 malware (150MB) obtained from the *National Cyber-Forensics and Training Alliance (NCFTA) Canada*[8].

Figure 12a depicts the runtime for clone detection methods for a window size ranging from 10 to 30 on *malware297*. Increasing the window size would increase the number of captured features. *ScalClone* captured 5, 5, and 10 features for a windows size of 10, 20, and 30, respectively. As explained in Section 4.2.4, a higher number of features results in a higher number of inexact clone groups. The chosen window sizes create 10, 10, and 45 inexact clone groups. Having a higher number of inexact clone groups increases the runtime of the *feature processor* step because we are storing more clone

[8]http://www.ncfta.ca

(a) Efficiency

(b) Feature processor scalability

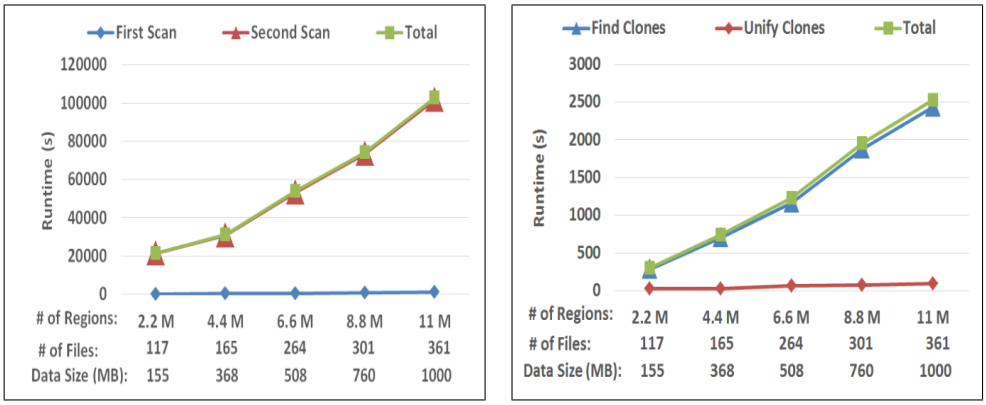


(c) Clone searcher scalability

Figure 12: Efficiency and scalability on *malware297*

groups into the database. Yet, it also increases the precision of the found clones. Figure 12a shows that the unification process runtime decreases with respect to the increase of window size, due to a smaller number of clones.

Figure 12b depicts the runtime for *feature processor* for the number of regions ranging from 150,000 to 750,000 extracted from *malware297*, with data size ranging from 27MB to 150MB. The *feature processor* makes two scans on the files. In the first scan, the processor extracts the significant features and filters out less significant ones. The second scan is responsible for storing the functions, regions, imports, constants, strings, features, and inexact clone groups into the database. The runtime increases from 1,157s to 9,860s in total as the number of regions increases from 150,000 to 750,000.



(a) Feature processor scalability (b) Clone searcher scalability

Figure 13: Scalability of *ScalClone* on *DLL1GB*

This figure suggests that the time complexity of the *feature processor* is linear with respect to the data size.

Figure 12c depicts the runtime of the *clone search* process for the same set of files discussed in *feature processor*. The target assembly file is a malware obtained from NCFTA Canada. The figure depicts that both the unification and clone searching steps yield linear time complexity. The runtime increases from 145s to 11,450s in total, as the number of regions increases from 150,000 to 750,000.

In the *zlib* benchmark, the window sizes are not fixed and depend on the source code clones sizes. But, in the *DLL18* and malware assortment, we could manually choose different window sizes to check their impact on the results to calculate the efficiency.

Scalability on *DLL1GB*: The fourth data set is an assortment of 1GB of DLL files obtained from an operating system. Figure 13a depicts the runtime for *feature processor* for the number of regions ranging from 2.2 million to 11 million, with data size ranging from 155MB to 1GB. The runtime increases from 21,401s to 102,605s in total as the number of regions increases to 11 million with 1GB of data. This figure also suggests that the time complexity of the *feature processor* is linear with respect to the data size.

Figure 13b depicts the runtime of the *clone search* process for the same set of files. The target assembly file is one of the DLL files in the data set. The figure depicts that both the unification and clone searching steps yield linear time complexity. The runtime increases from 308s to 2,525s in total

as the number of regions increases from 2.2 million to 11 million.

***BinClone* [1] Comparison**: We used the same dataset and configurations to compare the scalability of *ScalClone* and *BinClone* [1]. Figure 14 depicts the runtime for both methods on 10 to 70 malware files with a window size $w = 40$, and a minimum similarity $minS = 0.8$. The total processing time ranges from 35 to 980 seconds for *BinClone*, while *ScalClone* has a linear time complexity ranging from 16 to 92 seconds. This comparison suggests that adding a database on the backend can significantly improve the scalability of the clone search process.

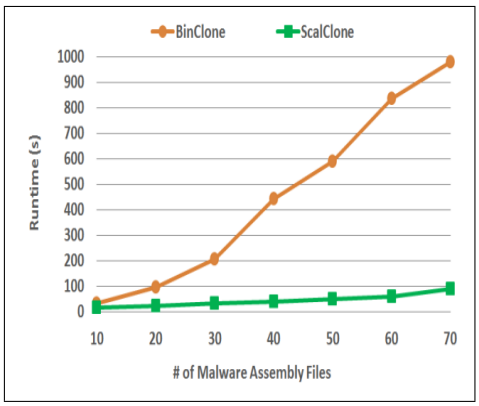


Figure 14: Scalability comparison between *ScalClone* and *BinClone*

7. Conclusions

In this paper, we present a scalable assembly code clone search system called *ScalClone*, which incorporates a database in the backend, in order to handle a large volume of assembly files. First, we present an effective assembly code clone search system with a high recall rate. Second, unlike the LSH approach employed in [4], our proposed clone detection methods are deterministic, an important property for malware analysis. Third, thanks to the well-designed database schema and operations, experimental results strongly suggest that our proposed clone search system is scalable to handle a large volume of assembly code. Fourth, experimental results also suggest that *ScalClone* can still effectively identify the clones with the presence of dead code insertion, instruction reordering, and compiler optimization. Finally, we present a system evaluation method for assembly code clone search,

with respect to the results of source code clone search. We demonstrate the feasibility of identifying high-quality clones from assembly code.

Acknowledgements

This research is supported by the NSERC DNDPJ 444877-12 and Defence Research and Development Canada (contract no. W7701-155902/001/QCL).

[1] M. R. Farhadi, B. C. M. Fung, P. Charland, M. Debbabi, Binclone: Detecting code clones in malware, in: Proceedings of the 8th IEEE International Conference on Software Security and Reliability (SERE), IEEE, 2014, pp. 78–87.

[2] P. Charland, B. C. M. Fung, M. R. Farhadi, Clone search for malicious code correlation, in: Proceedings of the NATO RTO Symposium on Information Assurance and Cyber Defense (IST-111), Koblenz, Germany, 2012, pp. 1.1–1.12.

[3] C. K. Roy, J. R. Cordy, A survey on software clone detection research, Tech. Rep. 541, Queen's University (September 2007).

[4] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, Z. Su, Detecting code clones in binary executables, in: Proceedings of the 18th International Symposium on Software Testing and Analysis, ACM, 2009, pp. 117–128.

[5] H. A. Basit, S. Jarzabek, Efficient token based clone detection with flexible tokenization, in: Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ACM, 2007, pp. 513–516.

[6] B. Hummel, E. Juergens, L. Heinemann, M. Conradt, Index-based code clone detection: incremental, distributed, scalable, in: Proceedings of the IEEE International Conference on Software Maintenance, IEEE, 2010, pp. 1–9.

[7] J. H. Johnson, Identifying redundancy in source code using fingerprints, in: Proceedings of the Conference of the Centre for Advanced Studies on Collaborative research: Software Engineering, Vol. 1, IBM Press, 1993, pp. 171–183.

[8] A. Marcus, J. I. Maletic, Identification of high-level concept clones in source code, in: Proceedings of the 16th Annual International Conference on Automated Software Engineering (AES), IEEE, 2001, pp. 107–114.

[9] R. Brixtel, M. Fontaine, B. Lesner, C. Bazin, R. Robbes, Language-independent clone detection applied to plagiarism detection, in: Proceedings of the 10th IEEE Working Conference on Source Code Analysis and Manipulation, IEEE, 2010, pp. 77–86.

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

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

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

[13] V. Wahler, D. Seipel, J. Wolff, G. Fischer, Clone detection in source code by frequent itemset techniques, in: Proceedings of the 4th IEEE International Workshop on Source Code Analysis and Manipulation, IEEE, 2004, pp. 128–135.

[14] J. Krinke, Identifying similar code with program dependence graphs, in: Proceedings of the 8th Working Conference on Reverse Engineering, IEEE, 2001, pp. 301–309.

[15] R. Komondoor, S. Horwitz, Using slicing to identify duplication in source code, Static Analysis (2001) 40–56.

[16] E. Balazinska, M.and Merlo, M. Dagenais, B. Lague, K. Kontogiannis, Measuring clone based reengineering opportunities, in: Proceedings of the 6th International Conference on Software Metrics Symposium, IEEE, 1999, pp. 292–303.

[17] I. Keivanloo, J. Rilling, P. Charland, Seclone-a hybrid approach to internet-scale real-time code clone search, in: Proceedings of the 19th

International Conference on Program Comprehension (ICPC), IEEE, 2011, pp. 223–224.

[18] J. Jang, D. Brumley, BitShred: Fast, scalable code reuse detection in binary code, Tech. Rep. CMU-CyLab-10-006, Carnegie Mellon University (2009).

[19] J. Jang, D. Brumley, S. Venkataraman, BitShred: Fast, scalable malware triage, Tech. Rep. CMU-Cylab-10-022, CMU (2010).

[20] A. Rahimian, P. Charland, S. Preda, M. Debbabi, RESource: a framework for online matching of assembly with open source code, in: Foundations and Practice of Security, Springer, 2013, pp. 211–226.

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

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

[23] A. Walenstein, M. Venable, M. Hayes, C. Thompson, A. Lakhotia, Exploiting similarity between variants to defeat malware, in: Proceedings of the BlackHat DC Conference, 2007.

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

[25] T. Dullien, E. Carrera, S. M. Eppler, S. Porst, Automated attacker correlation for malicious code, Tech. rep., DTIC Document (2010).

[26] E. Carrera, G. Erdélyi, Digital genome mapping–advanced binary malware analysis, in: Proceedings of the Virus Bulletin Conference, 2004, pp. 187–197.

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

[28] H. Flake, Structural comparison of executable objects, in: Proceedings of the International GI Workshop on Detection of Intrusions and Malware & Vulnerability Assessment, 2004, pp. 161–174.

[29] P. M. Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Kruegel, S. Zanero, Identifying dormant functionality in malware programs, in: Proceedings of the IEEE Symposium on Security and Privacy, IEEE, 2010, pp. 61–76.

[30] W. Jin, S. Chaki, C. Cohen, A. Gurfinkel, J. Havrilla, C. Hines, P. Narasimhan, Binary function clustering using semantic hashes, in: Proceedings of the 11th International Conference on Machine Learning and Applications (ICMLA), Vol. 1, IEEE, 2012, pp. 386–391.

[31] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, E. Kirda, Scalable, behavior-based malware clustering, in: Proceedings of the Network and IT Security Symposium (NDSS), Vol. 9, 2009, pp. 8–11.

[32] Z. Wang, K. Pierce, S. McFarling, BMAT: a binary matching tool for stale profile propagation, The Journal of Instruction-Level Parallelism 2 (2002) 2000.

[33] W. M. Khoo, A. Mycroft, R. Anderson, Rendezvous: a search engine for binary code, in: Proceedings of the 10th International Workshop on Mining Software Repositories (MSR), IEEE Press, 2013, pp. 329–338.

[34] V. Roussev, G. G. Richard, L. Marziale, Multi-resolution similarity hashing, digital investigation 4 (2007) 105–113.

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

[36] A. Andoni, P. Indyk, Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions, in: Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science, IEEE, 2006, pp. 459–468.