DRDC Tracking No: N14-0326-1311 DRDC Doc No: DRDC-RDDC-2014-P77

2014 Eighth International Conference on Software Security and Reliability

BinClone: Detecting Code Clones in Malware

Mohammad Reza Farhadi
Information Systems Engineering
Concordia University
Montreal, QC, Canada
Email: mo_farha@ciise.concordia.ca

Benjamin C. M. Fung
School of Information Studies
McGill University
Montreal, QC, Canada
Email: ben.fung@mcgill.ca

Philippe Charland
Mission Critical Cyber Security Section
Defence R&D Canada - Valcartier
Quebec, QC, Canada
Email: philippe.charland@drdc-rddc.gc.ca

Mourad Debbabi
Information Systems Engineering
Concordia University
Montreal, QC, Canada
Email: debbabi@ciise.concordia.ca

***Abstract*—To gain an in-depth understanding of the behaviour of a malware, reverse engineers have to disassemble the malware, analyze the resulting assembly code, and then archive the commented assembly code in a malware repository for future reference. In this paper, we have developed an assembly code clone detection system called *BinClone* to identify the code clone fragments from a collection of malware binaries with the following major contributions. First, we introduce two *deterministic* clone detection methods with the goals of improving the recall rate and facilitating malware analysis. Second, our methods allow malware analysts to discover both exact and inexact clones at different token normalization levels. Third, we evaluate our proposed clone detection methods on real-life malware binaries. To the best of our knowledge, this is the first work that studies the problem of assembly code clone detection for malware analysis.**

I. Introduction

Reverse engineering is a manual and time-consuming process, but it is often the primary step taken to gain an in-depth understanding of a piece of malware. To achieve a more efficient analysis, reverse engineers 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. Yet, the comparison process itself is also a challenging task, and the chances of identifying similar code fragments often depend on the experience and knowledge of the analyst. In this paper, we present an assembly code clone detection system to support malware analysis and evaluate its performance in terms of accuracy, efficiency, and scalability on the assembly code of real life binary and malware files.

The problem of assembly code clone detection for malware analysis is informally described as follows: Given a large repository of previously analyzed malware files and a new target file, the goal is to identify all the code fragments in the repository that are syntactically or semantically similar to the code fragments in the target malware file. The challenges associated with this problem 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 other fragments. Another alternative method is to perform a keyword search using the RESource IDA Pro plug-in [28]. Although essential, the keyword search capability is insufficient for assembly code clone detection, as many code fragments do not contain any uniquely identifying keywords.

Large code volume: The size of an assembly file can range from a couple of kilobytes to over dozens of megabytes of textual data. The efficiency of a clone detection method refers to the period of time required to identify all the clones. Scalability refers to its capability to handle a large collection of assembly code.

Deterministic clone results: To support effective malware analysis, the clone detection method must be deterministic, i.e., the identified clone detection results have to be repeatable given the same search query and malware repository. Otherwise, it will be very difficult for a reverse engineer to draw any conclusions on the clone results.

Contributions: Sæbjørnsen et al. [30] presented a method to identify 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 this approach shows some encouraging results in identifying clone pairs that are not exactly identical, its assumption on the uniform distribution of vectors may not hold as the number of features (i.e., dimensions) increases, resulting in false negatives. In this paper, we develop an assembly code clone detection framework based on [30] and made significant extensions in different components to support the special needs of clone detection for malware analysis. The contributions of this paper are summarized as follows:

- *Improved inexact clone detection.* We propose two new inexact clone detection methods to improve the recall rate and to improve the efficiency by employing a filtering process. Also, our technique is robust against clones that

978-1-4799-4296-1/14 $31.00 © 2014 IEEE
DOI 10.1109/SERE.2014.21

IEEE computer society

have reordered instruction caused by different compiler optimization methods.

- *Flexible normalization procedure.* We present a generic method to allow malware analysts to discover both exact and inexact clones at different normalization levels of assembly tokens, namely, memory references, registers, and constant values. Experimental results suggest that normalizations can help improve the recall when identifying Type I and II clones.
- *Support for malware analysis.* The clone detection method has to be deterministic in order to support malware analysis. Thus, any clone detection method that employs randomization, such as the LSH approach used in [30], does not satisfy this requirement. For this reason, our proposed inexact clone detection methods are deterministic. Finally, we evaluate our proposed methods on real-life malware binaries obtained from the National Cyber-Forensics and Training Alliance (NCFTA) Canada[1]. To the best of our knowledge, this is the first work that studies the problem of assembly code clone detection for malware analysis. Experimental results on real life malware binaries suggest that our proposed clone detection algorithm is effective.

The rest of this paper is organized as follows: Section II provides related work and background information on clone detection, followed by a formal definition of the clone detection problem in Section III. The proposed clone detection framework is described in Section IV, and its performance evaluation is presented in Section V. Section VI concludes the paper.

II. Related Work

Most of the existing source code clone detection techniques are not directly applicable to assembly code. 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 consists of a large collection of instructions that share a nearly identical format. Applying the existing source code clone detection methods such as program dependency graph (PDG) [24][22], text [18][25], tree [11][32], token [2][14], similarity distance [5], and metric-based approaches [23][26] on assembly code may result in many false positives (i.e., low precision). Also, as most malware are not written in Java, Java bytecode clone detection methods [20][27][13] are not directly applicable for malware analysis. For example, Java bytecode does not consider registers.

The following definitions of clone types are commonly used in the literature [29] of source code clone detection. Type I, II, and III are known as textual clones, while Type IV clones are known as semantic clones. We utilize the same clones types 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 identifiers, literals, types, layouts, and comments.

Type III: Copied fragments with further modifications. Statements can be changed, added, or removed, in addition to variations in identifiers, literals, types, layouts, and comments.

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

The clone detection methods on assembly code can be broadly categorized as follows:

Text-based approach: Jang et al. [15][16] 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 requirement of any comparison system.

Token-based approach: Schulman [31] proposed 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. Karim et al. [19] addressed the problem of classifying new malware into existing malware families whose individual entries share common code. Walenstein et al. [33] further extended this approach to match assembly code by comparing n*-grams* and n*-perms* extracted from disassembled binaries that have been unpacked and unencrypted. A vector model for the comparison of entire binaries is used but is ineffective, unless most of the binary is a clone of another previously analyzed binary.

Metric-based approach: Bruschi et al. [6] presented a technique to normalize assembly code to detect 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 that can be used to measure the similarity of samples. Sæbjørnsen et al. [30] 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.

Structural-based approach: Dullien et al. [10] presented the research results on executable code comparison for attacker correlation. They implemented a system that can identify code similarities in executable files. Dullien also developed a software tool called *BinDiff*[2] that compares binary files using a control flowgraph. It parses and extracts features of every function, and then iteratively matches its callers and callees. The objective of *BinDiff* is to analyze software patches. Carrera and Erdelyi [7] 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. [4] 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 [12] presented a method that constructs an isomorphism between the groups of functions that are used into two different versions of the same binary.

BinCrowd[3] creates a repository that stores the analyzed assembly code together with their function names, comments,

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

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

[3]http://www.zynamics.com/bincrowd.html

or other related information. This tool employs the *BinDiff* algorithm to discover known functions in other disassembled files. Christodorescu et al. [8] 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. One drawback of this approach is that it decides that some portions of a given executable resemble another portion of another executable without considering whether or not they are the same. Anju et al. [1] proposed a method for malware detection based on control flow graph optimization. They defined architecture for detecting malicious patterns in executable files.

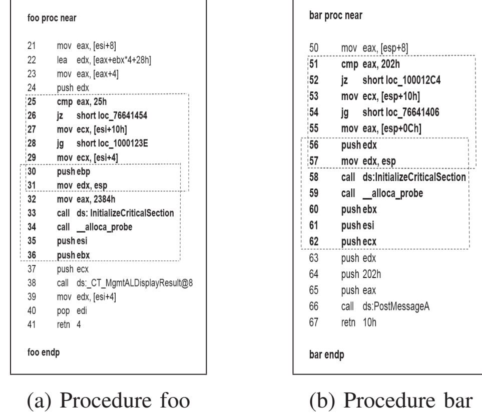
Behavioral-based approach: Comparetti et al. [9] developed a system called *REANIMATOR* that allows the identification of dormant functionalities in malware. They exploit the fact that a dynamic malware analysis captures malware execution and reports its behavior. This approach can be useful if one wants to match different code portions that are semantically identical but syntactically different. Jin et al. [17] 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 [3] proposed a method to group malware by their behaviour, but this problem is different from assembly clone detection.

Hybrid approach: Wang et al. [34] presented a tool called *BMAT* that creates mappings between old and new versions of binaries. This tool is used to generate profile information of applications and illustrates how to propagate stale profiles from an old to a new version. Khoo et al. [21] developed a search engine that allows searching for binary code against some existing open source and code repositories. First, they tokenized assembly functions and then extracted some features to build a query term. 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.

III. PROBLEM DEFINITION

We first provide an informal description of the assembly code clone detection problem followed by a formal problem description and an example. Given a large collection of previously analyzed assembly files and a specific target assembly file or code fragment, a user would like to identify all the code fragments in the previously analyzed assembly files that are syntactically or semantically similar to the target assembly file or code fragment.

Let $A = \{A_1, \ldots, A_n\}$ be a collection of previously analyzed assembly files, where each assembly file A_i consists of m lines of assembly code, denoted by $f[1 : m]$. In the rest of this section, we assume the assembly code has been normalized according to Section IV. 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 $1 \leq a \leq b \leq m$.



(a) Procedure foo (b) Procedure bar

Fig. 1: Sample Procedures

$|f[a : b]|$ denotes the number of lines of assembly code in $f[a : b]$. We define two notions of clones as follows: Intuitively, two code fragments are an *exact clone pair* if they have the same sequence of assembly instructions. Two code fragments that share similar instructions with respect to the mnemonics and operands are considered as an *inexact clone pair*.

Definition 3.1 (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 types appearing in the line of assembly code instruction. ■

Definition 3.2 (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. 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, for malware analysis, a reverse engineer might sometimes want to *efficiently* identify only the exact clones. The *problem of assembly code clone detection* is to identify *all* exact and inexact clones from the collection of assembly files A.

Example 1: Suppose the collection of assembly code A contains only two procedures as shown in Figure 1. The code fragment $f[25, 31]$ in foo *and the code fragment $f[51, 57]$ in* bar *are an exact clone pair. Also, the code fragment $f[30, 36]$ in* foo *and the code fragment $f[56, 62]$ in* bar *are an inexact clone pair.* ■

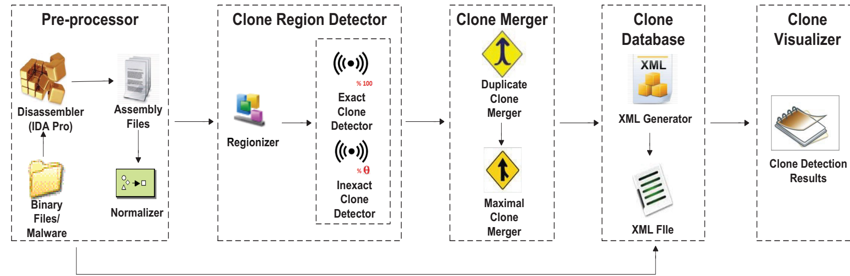
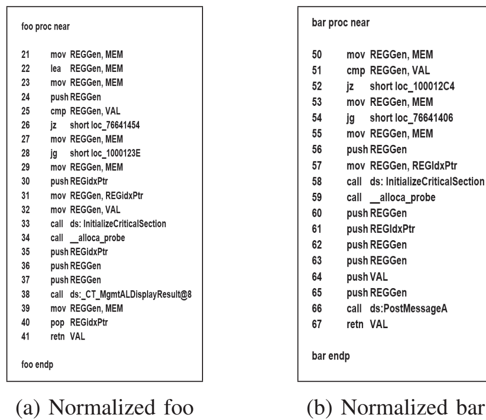


Fig. 3: System Architecture



(a) Normalized foo (b) Normalized bar

Fig. 2: Normalized Procedures

IV. Clone Detection System

The proposed clone detection system consists of five major components, namely the pre-processor, clone region detector, clone merger, clone database, and clone visualizer. Figure 3 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. The clone merger then combines consecutive clone regions into larger clones. Afterwards, 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.

A. Pre-processor

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

1) Disassembler: This step disassembles the input binary files into the collection of assembly files A using a disassembler such as IDA Pro. Each assembly file $A_f \in A$ contains a set of functions. Each function contains a sequence of assembly 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, namely *memory reference*, *register reference* and *constant values*.

2) 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 selected by the user. For constant values, the user has the possibility to generalize them to VAL_x, where x is an index number, or to VAL, which simply ignores the exact constant value. For registers, the user can generalize them according to the normalization hierarchy depicted in Figure 4. The top-most level *REG* generalizes all registers disregarding 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). Finally, the bottom level breaks down the General Registers into 3 groups by size, namely 32-, 16-, and 8-bit registers.

Figures 2a and 2b show the normalized versions of *foo* and *bar* using the second level of the hierarchy.

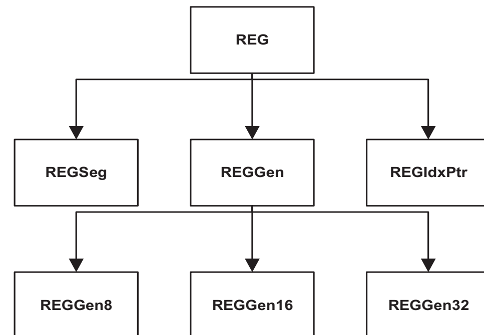


Fig. 4: Normalization Hierarchy for Registers

B. Clone Region Detector

The clone region detector consists of three steps. The first step partitions each function into an array of regions. The

second and third steps respectively identifies exact and inexact clones, among the regions created in the first step.

1) Regionizer: Each function is partitioned into an array of overlapping regions using a sliding window with a size of at most w statements, where w is a user-specified threshold. Figure 5 shows the extracted regions of the normalized procedure *foo* in Figure 2a with $w = 15$.

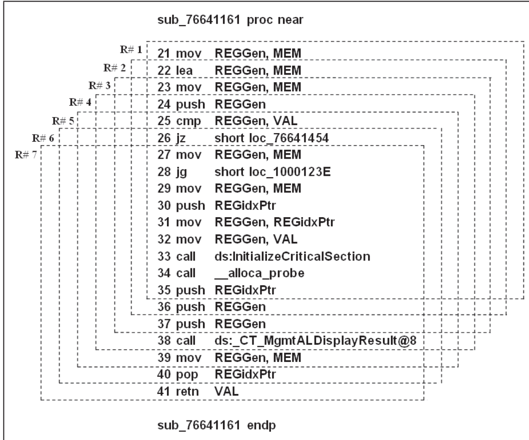


Fig. 5: Regionization for $w = 15$

2) 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. 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 naive approach is too computationally expensive with a complexity of $O(n^2)$, where n is the total number of regions. Thus, we used a hashing approach with linear complexity and the same accuracy. 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 negatives. The process requires only one scan of the 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.

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.

Algorithm 1: Exact Clone Detector

input : set of regions R
output: set of exact clone pairs EC

```
1  begin
2      H ← ∅;
3      EC ← ∅;
4      foreach region r ∈ R do
5          v ← hash(r);
6          H(v) ← H(v) ∪ {r};
7      foreach H(v) ∈ H do
8          for i = 0 → |H(v)| do
9              for j = i + 1 → |H(v)| do
10                 EC ← EC ∪ {(r_i, r_j)};
11     return EC;
```

The feature vectors are constructed based on four groups of features from the assembly instructions [30]. 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. Finally, the last group includes all combinations of the first two operands.

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

Sequential Feature Selection Method: Algorithm 2 provides an overview of the sequential feature selection method for inexact clone detection.

1) **Compute medians**: This step computes the median of each feature on all regions. 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 and decrease the false positive error rate.
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 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.

Algorithm 2: Sequential Feature Selection

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

1 $SBSize \leftarrow sub-vectors'size$; $Binary \leftarrow \emptyset$;
2 $H_k \leftarrow \emptyset$;
3 $M \leftarrow ComputeMedians(F)$; /* Step1 */
4 **foreach** $m \in M$ **do** /* Step2 */
5   **if** $m = 0$ **then**
6     $M \leftarrow M - m$;
7 **foreach** $r \in R$ **do** /* Step3 */
8   **for** $k = 0 \rightarrow length(M)$ **do**
9   **if** $F[k] \geq M[k]$ **then**
10       $Binary[k] \leftarrow 1$;
11   **else**
12       $Binary[k] \leftarrow 0$;
13 **foreach** $r \in R$ **do** /* Step4 */
14   **for** $i = 0 \rightarrow length(Binary) - SBSize + 1$ **do**
15   **for** $j = 0 \rightarrow SBSize$ **do**
16       $sub - vector_i[j] \leftarrow Binary[i + j]$;
17 **foreach** $r \in R$ **do** /* Step5 */
18   **foreach** $k = 0 \rightarrow Numberofsub - vectors$ **do**
19     $H_k \leftarrow$ Compute the decimal number;
20     $R\prime \leftarrow$ find other regions with the same hash value
21   **foreach** $r\prime \in R\prime$ **do**
22   **if** *r and r′ occurred more than the minS threshold* **then**
23       $IC \leftarrow IC \cup \{(r, r\prime)\}$;
24 **return** IC;

Example 2: Figure 6 shows a collection of features generated from a dataset after the filtering process. For simplicity, only a small set of features is shown here. The dashed rectangles show the sub-vectors of the feature vector with a user-defined sub-vector of size 5. Let the user-defined sub-vector size be 5. There are thus $n - 5 + 1$ sub-vectors for n extracted features after the filtering process and $2^5 = 32$ possible hash values (decimal numbers) for each sub-vector, resulting in 32 entries in each associated inexact hash table. Step 5 maps the regions into these hash tables by computing the decimal number of their binary vectors. ■

Two-Combination Method: 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 size, all possible two-combinations of the remaining features after the filtering process are constructed. Each two-combination vector acts as a sub-vector. Then, each feature vector is mapped into its sub-vectors. Sub-vectors are the same size as two-combinations, which is equal to 2. In this case, the user does not have the flexibility to choose the size of sub-vectors.

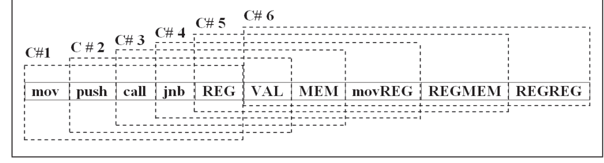


Fig. 6: Step 3 - SFS Inexact Detection Method with $SBSize = 5$

Example 3: Figure 7 shows all possible two-combinations of the features in Figure 6, 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. ■

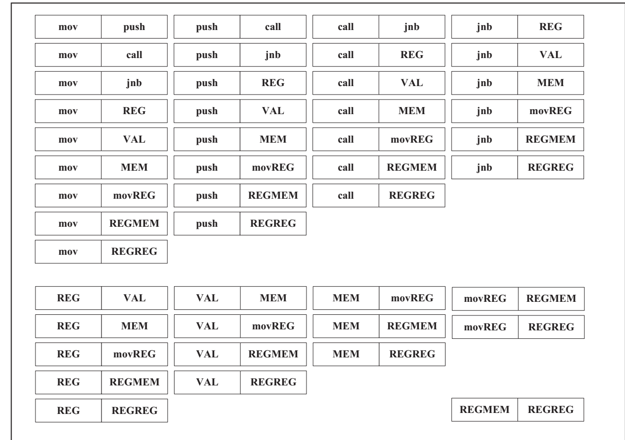


Fig. 7: Step 3 - TC Inexact Detection Method

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 inexact clone detection.

The sequential feature selection 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 sequential feature selection method is a subset of the sub-vectors generated by the two-combination method. As a result, the sequential feature selection 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 in Section V also support this observation.

```
1   mov   REGGen, MEM
2   lea   REGGen, MEM
3   mov   REGGen, MEM
4   push  REGGen
5   cmp   REGGen, VAL
6   jz    MEM
7   mov   REGGen, MEM
8   jg    MEM
9   mov   REGGen, MEM
10  push  REGIdxPtr
11  push  REGGen
12  mov   REGGen, MEM
13  mov   REGGen, MEM
14  lea   REGGen, MEM
```

Fig. 8: Duplicate Clone Merger with $w = 10$ and overlapped size of 0.6

```
50  mov   REGGen, MEM      105  mov   REGGen, MEM
51  cmp   REGGen, VAL      106  cmp   REGGen, VAL
52  jz    MEM              107  jz    MEM
53  mov   REGGen, MEM      108  mov   REGGen, MEM
54  jg    MEM              109  jg    MEM
55  mov   REGGen, MEM      110  mov   REGGen, MEM
56  push  REGGen           111  push  REGGen
57  mov   REGGen, MEM      112  mov   REGGen, MEM
58  call  MEM              113  call  MEM
59  call  MEM              114  call  MEM
60  push  REGGen           115  push  REGGen
61  push  REGIdxPtr        116  mov   REGGen, VAL
62  push  REGGen           117  push  REGGen
63  push  REGGen           118  call  MEM
64  push  VAL              119  push  REGGen
65  push  REGGen           120  mov   REGGen, VAL
66  call  MEM              121  pop   REGIdxPtr
67  retn  VAL              122  retn  VAL
```

Fig. 9: Maximal Clone Merger with $w = 5$

C. Clone Merger

1) Duplicate Clone Merger: The inexact clone detector may misclassify two consecutive regions to be a clone. This step is intended to remove the clones that are highly overlapping consecutive regions. A fixed overlapping threshold of 50% is defined that indicates the fraction of allowed overlapped instructions of two consecutive regions that can still be considered as a clone pair. In other words, the consecutive regions with half part overlapped instructions would be discarded.

Example 4: Figure 8 provides an example for this step. It shows two consecutive regions with a window of size 10 and an overlapping ratio of 60%. Since the overlapping ratio is above %50 , the clone pair is discarded. ■

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 process 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 overlapping clones in lines 4-5.

Example 5: Suppose clone pairs c and c' are a pair of regions $\{A, B\}$ and $\{A', B'\}$, respectively. Two clones are overlapping if each of their regions shares some instructions. Hence, c and c' are overlapping cloned pairs if $\{A, A'\}$ and $\{B, B'\}$ have overlapping instructions. ■

Example 6: Figure 9 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 6 clone pairs. Since all 6 clone pairs are consecutive, they are merged into one clone pair, as indicated by the dashed rectangles. ■

D. Clone Database

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*.

Algorithm 3: Maximal Clone Merger

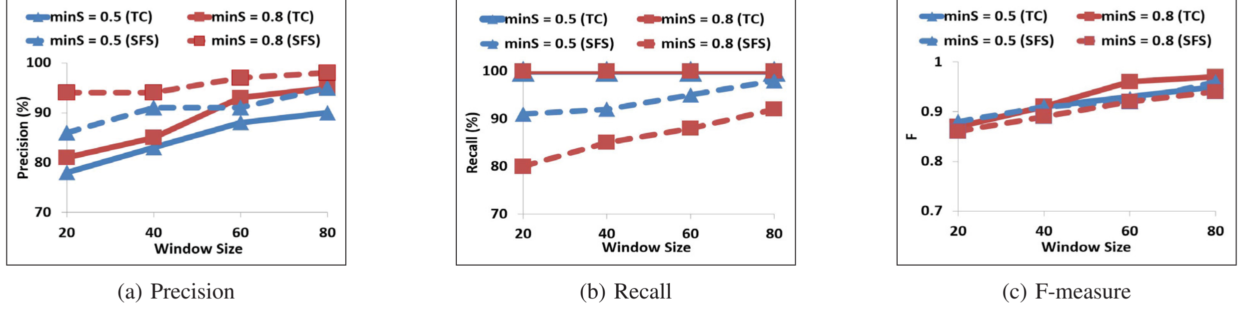
input : set of clone pairs CP
output: set of maximal merged clone pairs MC

```
1 begin
2   MC ← ∅;
3   foreach clone pair c and c′ ∈ CP do
4     if overlap(region A ∈ c, region A′ ∈ c′) and
        overlap(region B ∈ c, region B′ ∈ c′) then
5       CP ← merge(c, c′);
6   MC ← CP;
7   return MC;
```

- The *parameters* node stores the user-specified parameters, such as the window size w, minimal similarity threshold $minS$, and normalization level.
- 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.
- The *clone_files* node stores the clone detection results. Specifically, the *clone_files* node stores a list of clone files, in which each *clone_file* stores a list of *clone_pairs*.
- The *token_references* node stores the token indexes. Specifically, the *token_references* node stores three lists of tokens, namely *constants*, *strings*, and *imports*, with their locations indicated by *fileID*s and line numbers.

E. Clone 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. This set consists of *assembly folder path*, *XML report destination path*, *window size*, minimal similarity $minS$, *register normalization level*, *choice of inexact detection method*, and *sub-vector size*, if the inexact detection sequential feature selection method is selected. Next, the program shows



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

Fig. 10: Accuracy (DLL files)

Name	Type	Size	# of Functions	# of LOC
Zeus	Trojan Horse	9 MB	45,954	594,153
Blaster	Worm	70 KB	13	2,642

TABLE I: Malware Specifications

Window Size:	20	40	60	80
# of Exact Clones	18,010	17,225	17,162	16,971
# of Inexact Clones (SFS)	266,335	272,008	274,346	759,953
# of Inexact Clones (TC)	285,132	441,575	736,396	1,053,801

TABLE II: Number of Clones (Malware Assortment)

all pairs of clone files. A user can click on a file pair to see a list of clone pairs between the two files.

V. 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. The program was written in Visual C++. 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 converted into 18 assembly files using IDA Pro. 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. Zeus is also known as the king of bank fraud trojan viruses, having been used by thousands of criminals to scam from banking customers around the world for years. *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. The intent of choosing these two well-known malware is to show that malware share a large number of code clones. Table I shows some basic information on the two disassembled malware. The second dataset is an assortment of 70 malware obtained from NCFTA Canada. The files were disassembled and their total size is over 10 MB.

A. 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:

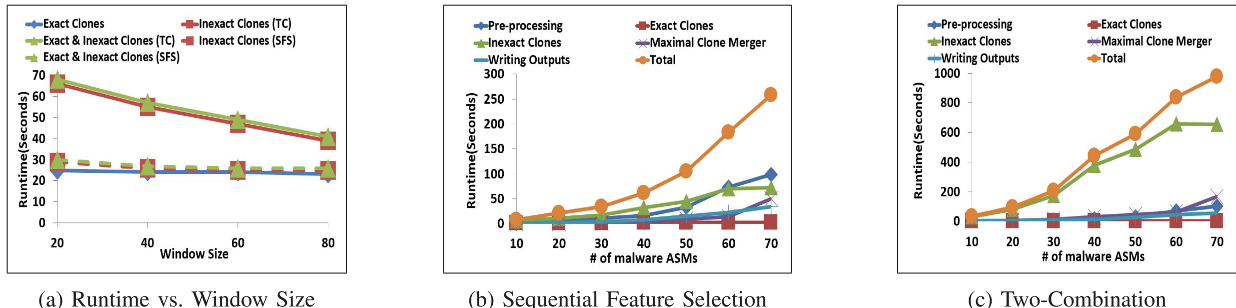
$$Precision(Solution, Result) = \frac{n_{ij}}{|Result|} \quad (1)$$

$$Recall(Solution, Result) = \frac{n_{ij}}{|Solution|} \quad (2)$$

$$F(Solution, Result) = \frac{2 \times Recall \times Precision}{Recall + Precision} \quad (3)$$

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*. To compute the precision of the exact detection, we observed if the found code clones were exactly the same with respect to Definition 3.1. The same strategy was applied to compute the precision of inexact detection methods with respect to the chosen minimum similarity threshold $minS$. 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.

Figure 10 shows the precision, recall, and F-measure for minimum similarity threshold $minS = 0.5$ and $minS = 0.8$ using the assortment of DLL files. Both the sequential feature selection and the two-combination inexact detection methods are evaluated. Experimental results show a better precision for the sequential feature selection when compared with the two-combination method for inexact clone detection. By considering the fact that the sequential feature selection has fewer sub-vectors, Figure 10a 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, as explained in Section IV-B3. The precision and F-measure are consistently above 75% for both inexact detection methods. The recall is above 80% for the sequential feature selection



(a) Runtime vs. Window Size

(b) Sequential Feature Selection

(c) Two-Combination

Fig. 11: Efficiency and Scalability

and 100% for the two-combination method, suggesting that the clone detection methods are effective.

To evaluate the precision of the second dataset (*Zeus* and *Blaster*), the first 10 regions were selected from each malware. Then, for each selected region, the proposed clone detection methods were used to search for its clones in the rest of the code. Each of the identified clone in the result was manually examined in order to compute the precision for both exact and inexact code clones. To compute the precision of the exact detection, we observed if the found code clones are exactly the same with respect to Definition 3.1. The same strategy was applied to compute the precision of inexact detection methods with respect to the chosen minimum similarity threshold $minS$. With 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.

We also evaluated the number of exact and inexact clones identified by our method in the third dataset (malware collection) for different window sizes for both the sequential feature selection and two-combination inexact detection methods. Table II shows that there is a large number of exact and inexact clones in malware. The results suggest that malware programmers reuse code at both regional and functional levels. Also, the two-combination method (TC) can identify more clones than the sequential feature selection method (SFS).

B. Efficiency and Scalability

Figure 11a depicts the runtime for both exact and inexact clone detection methods for a window size ranging from 20 to 80 using the malware assortment dataset. The process took 26 to 30 seconds when the sequential feature selection inexact clone detection method was used and 41 to 68 seconds for the two-combination inexact clone detection. In general, runtime decreases as the window size increases, as fewer regions results in fewer clones. Figure 11b illustrates the runtime of the sequential feature selection method using 10 to 70 malware files with window size $w = 40$, and minimum similarity $minS = 0.8$. The total processing time for the sample malware assortment ranges from 8 to 258 seconds. Figure 11c shows the runtime for the same dataset and settings for the two-combination inexact detection method. The total processing time ranges from 35 to 980 seconds. As mentioned in Section IV-B3, the sequential feature selection method performs better in terms of scalability, as it has fewer number of inexact sub-vectors.

C. Comparison with the LSH Approach

Sæbjørnsen et al. [30] presented an inexact clone detection method to identify inexact clone pairs by 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 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:

$$lsh(v) = (h_1(v), h_2(v), \ldots, h_m(v)) \tag{4}$$

In brief, the LSH method splits a vector space into 2^m subspaces by m base vectors. These base vectors are chosen randomly 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 parameters selection, which is challenging in large dimension 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, as it will be very difficult for a reverse engineer to produce a consistent malware analysis. 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, while the second parameter is the subspaces dimensions.

VI. Conclusions

This paper has presented some significant extensions based on the work of Sæbjørnsen et al.
[30]. First, we proposed two efficient and effective inexact clone detection methods capable of finding Type III clones. Experimental results suggest that the two-combination inexact detection method can eliminate all false negatives. Second,

unlike the LSH approach employed in [30], our proposed clone detection methods are deterministic, an important property for malware analysis as specified by reverse engineers. Third, we implemented a flexible normalization scheme to normalize assembly code instructions so that clone detection can be performed at different levels, depending on the purpose, to detect Type II and III clones. Experimental results on real life data suggest that our implemented system can effectively identify exact and inexact clones in assembly code.

VII. Acknowledgements

This research is supported by Defence Research and Development Canada (DRDC).

References

[1] 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.

[2] H. A. Basit and 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*, pages 513–516. ACM, 2007.

[3] Ulrich Bayer, Paolo Milani Comparetti, Clemens Hlauschek, Christopher Kruegel, and Engin Kirda. Scalable, behavior-based malware clustering. In *Proceedings of the Network and IT Security Symposium (NDSS)*, volume 9, pages 8–11, 2009.

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

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

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

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

[8] M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *Proceedings of the 1st India software engineering conference*, pages 5–14. ACM, 2008.

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

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

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

[12] H. Flake. Structural comparison of executable objects. In *Proceedings 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.

[13] Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song. Juxtapp: A scalable system for detecting code reuse among android applications. In *Proceedings of the 9th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, pages 62–81, 2012.

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

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

[16] 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.

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

[18] 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*, volume 1, pages 171–183. IBM Press, 1993.

[19] 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.

[20] I. Keivanloo, C. K. Roy, and J. Rilling. Java bytecode clone detection via relaxation on code fingerprint and semantic web reasoning. In *Software Clones (IWSC), 2012 6th International Workshop on*, pages 36–42. IEEE, 2012.

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

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

[23] 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.

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

[25] A. Marcus and 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)*, pages 107–114. IEEE, 2001.

[26] J. Mayrand, C. Leblanc, and 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*, pages 244–253. IEEE, 1996.

[27] L. Quesada, F. Berzal, and J. C. Cubero. Jsimil-a java bytecode clone detector. In *ICSOFT (2)*, pages 333–336, 2010.

[28] A. Rahimian, P. Charland, S. Preda, and M. Debbabi. Resource: a framework for online matching of assembly with open source code. In *Foundations and Practice of Security*, pages 211–226. Springer, 2013.

[29] C. K. Roy and J. R. Cordy. A survey on software clone detection research. Technical Report 541, Queen's University, September 2007.

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

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

[32] V. Wahler, D. Seipel, J. Wolff, and 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*, pages 128–135. IEEE, 2004.

[33] A. Walenstein, M. Venable, M. Hayes, C. Thompson, and A. Lakhotia. Exploiting similarity between variants to defeat malware. In *Proceedings of the BlackHat DC Conference*, 2007.

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