Clone Search for Malicious Code Correlation

Philippe Charland
Mission Critical Cyber Security Section
Defence Research and Development Canada – Valcartier
2459 Pie-XI Blvd North, Quebec, QC
Canada, G3J 1X5

philippe.charland@drdc-rddc.gc.ca

Benjamin C. M. Fung **Mohammad Reza Farhadi**
Concordia Institute for Information Systems Engineering
Concordia University
1455 de Maisonneuve Blvd. West, Montreal, QC
Canada, H3G 1M8

{fung, mo_farha}@ciise.concordia.ca

ABSTRACT

With the revolution in information technology, the dependence of the NATO countries on their information systems continues to grow. However, this represents a point of vulnerability, as these systems are exposed to malicious software (malware). Understanding malware to mitigate it requires software reverse engineering, but this is a manually intensive and time-consuming process. The learning curve to master it is quite steep and with today's proliferation of malware, this results in the very few available reverse engineers being quickly saturated. This article presents the research results on code clone search to accelerate the reverse engineering process. As developing stealthy and persistent malware requires a high degree of technical complexity, it is quite common for code fragments to be reused between different malware. The objective is thus to use code clone search to correlate previously analyzed with new malware to automatically identify the similarities between them and thereby, the code fragments they share. This would prevent reverse engineers from reanalyzing the code fragments of a new malware, which have already been analyzed in a previous context.

1.0 INTRODUCTION

The revolution in information technology is resulting in a growing dependence on information and communication systems and is a point of vulnerability for NATO countries. While information systems-based assets confer a distinct advantage for NATO militaries, these militaries are also vulnerable if adversaries interfere with these assets. Unfortunately, the technology required to disrupt and damage an information system is far less sophisticated and expensive than the amount of investment required to create the system. Cyber attacks offer an adversary maximum anonymity and a low risk of personal injury. The infrastructure required to conduct such attacks is relatively small, which makes this type of operation extremely attractive. In the past years, the overall sophistication, volume, and degree of coordination of these attacks have increased, which means that there will be a continuing demand for improved protection and countermeasures.

It is a common scenario that the only piece of evidence of a successful cyber attack is the malicious executable code itself. Analyzing malicious code (malware) requires software reverse engineering, as

malware source code is unavailable most of the time. Software reverse engineering is a manually intensive and time-consuming process, whose objective is to determine the functionality of a program. It consists in taking a program's executable binary, translating it into assembly code, and then manually analyzing the resulting assembly code. Most of the steps involve translating assembly code into a series of abstractions that represent the overall flow of a program to determine its functionality. The learning curve to master reverse engineering is quite steep and once mastered, the process is hindered when a program is obfuscated by anti-reversing techniques or actively tries to avoid detection, as most malware do. The demanding requirements of reverse engineering, combined with today's proliferation of malware, have resulted in the very few available reverse engineers being quickly saturated.

During the last few years, the sophistication of malware has evolved considerably. While it used to consist of small programs written mostly in assembly which spread by infecting other executable files, today's malware is written using high-level languages, comes in many forms (e.g., botnets, rootkits, malicious document files), and each new version of a malware (i.e., variant) improves on the previous one, by adding new capabilities and fixing bugs. As developing stealthy and persistent malware requires a high degree of technical complexity, it is quite common for code fragments to be reused between different malware.

The fact that malware authors exchange source code among them, have adopted a versioning approach, and use evasion techniques to bypass antivirus detection have resulted in a proliferation of malware. Reverse engineers should thus leverage the code reuse in the production of malware and be able to correlate different malware to identify the similarities between them and thereby, the code fragments they share. This would prevent reverse engineers from reanalyzing the code fragments of a new malware, which have already been analyzed in a previous context. A direct comparison (i.e., syntactic) of malware variants would be fruitless, as different compiler settings can be used to generate vastly different executable code from identical source code input. To address this problem, the present article applies clone detection and search to identify the code fragments shared between different malware to reduce redundant analysis efforts. The remainder of this article is organized as follows: Section 2 provides background information on clone detection, followed by a formal definition of the clone search problem in Section 3. The proposed clone search framework is described in Section 4 and its evaluation is presented in Section 5. Finally, Section 6 discusses the conclusion and future work.

2.0 BACKGROUND

Clone detection is a technique to identify duplicate code fragments in a code base. Traditionally, it has been used to decrease code size by consolidating it and thus, facilitate program comprehension and software maintenance. This need stems from the fact that reusing code fragments by copying and pasting them, with or without minor modifications, is a common scenario in software development that can be detrimental to software maintenance and evolution. For example, if a bug is found in a code fragment, then all similar code fragments must also be verified for the presence of this bug.

As clone detection is an important problem, it has been studied extensively and numerous clone detection techniques exist. Depending on the code level analysis used, they can be classified within the following categories: text-based [10, 16, 9], token-based [2, 11, 3, 8], tree-based [4, 22, 6], metrics-based [13, 17], and graph-based [14, 12, 15]. While most existing clone detection techniques operate on source code, clone detection has also been applied to binary code, since source code is not always available, as in the case of commercial off-the-shelf (COTS) software. One important application of clone detection on binary code is the detection of copyright infringements. For example, closed source software should not contain open source code released under the GNU General Public License (GPL). The proposed approach in this article

operates on binary code. But before describing it in detail, a brief introduction on the clone detection terminology is provided.

A code fragment is any sequence of code lines, with or without comments, at any granularity level (e.g., function, code block) [21]. A code fragment is a clone of another code fragment if they are similar according to a given definition of similarity [21]. In clone detection, code fragments can be similar based on their program text (textual similarity) or functionality (functional similarity). Two similar code fragments form a clone pair and several clone pairs form a clone cluster. In the literature, the following definitions of the different clone types are commonly used [20]:

- Textual Similarity
  - **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, layout, and comments.
  - **Type III:** Copied fragments with further modifications. Statements can be changed, added, or removed, in addition to variations in identifiers, literals, types, layout, and comments.
- Functional Similarity
  - **Type IV:** Code fragments which perform the same computation, but implemented using different syntactic variants. These are also referred as semantic clones.

3.0 ASSEMBLY CODE CLONE SEARCH

As previously mentioned, the objective of clone detection is to identify all the highly similar code fragments within a code base which, in the worst case, might involve the comparison of every code fragment pair. But given a collection of previously analyzed assembly files and a target assembly code fragment, such as in the case of malware analysis, the objective is not to identify all the duplicate code fragments. It is only to identify all the code fragments in the previously analyzed assembly files that are syntactically or semantically similar to the target assembly code fragment. This problem, known as assembly code clone search, is formally defined next.

Let $A = \{A_1,...,A_n\}$ be a collection of previously analyzed assembly files, where each assembly file A_i consists of a sequence of assembly code instructions $<c_1, \ldots, c_m>$. A code fragment f in an assembly file A_i refers to a subsequence of assembly code instructions $f = <c_i, \ldots, c_j>$ in A_i, where $1 \leq i \leq j \leq m$. Let $t = <t_1, \ldots, t_k>$ be the user-specified target code fragment. Let $sim(f_x, f_y)$ be a function that measures the similarity between two code fragments f_x and f_y. Let $minS$ be a user-specified minimum similarity threshold. The problem of assembly code clone search is to identify all matched code fragments M where every code fragment $f_x \in M$ satisfies the following conditions:

1. $\exists A_i \in A \mid f_x A_i$, i.e., the code fragment f_x is a subsequence of some assembly file A_i.

2. $sim(f_x, t) \leq minS$, i.e., the similarity between the code fragment f_x and the target code fragment t is within the threshold $minS$.

4.0 ASSEMBLY CODE CLONE SEARCH FRAMEWORK

4.1 Framework Overview

The code clone search prototype system developed in the context of this research work implements an improved version of the code clone detection framework proposed by Saebjornsen et al. [18]. Figure 1 provides an overview of its nine processes. A high-level description of each of them is first provided, followed by a detailed description of the normalization and inexact clone detection processes.

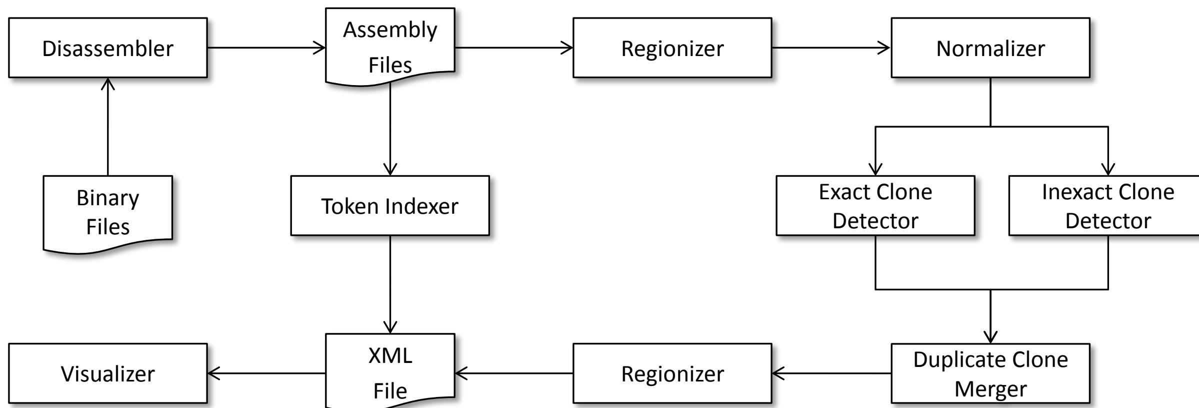


Figure 1: Code clone search process overview (extended from Saebjornsen et al. [18])

1. **Disassembler:** The first step is to disassemble the input binaries into assembly files using a disassembler, such as IDA Pro [7].
2. **Regionizer:** The second step consists of identifying all the functions for each assembly file. Then, each function is partitioned into an array of overlapping regions with a size of at most w instructions, using a sliding window with a step size of s, where w and s are user-specified parameters. Figure 2 below shows an example.

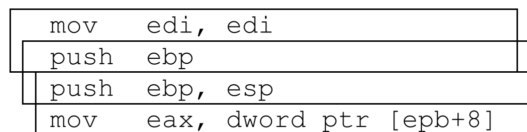


Figure 2: Regionizer with a window size of 2 and a stride of 1

3. **Normalizer:** The third step normalizes constants, memory addresses, and registers in each region to facilitate their comparison in the subsequent clone detection process. Section 4.2 illustrates the improvement made to the original normalization process.
4. **Exact clone detector:** A *clone pair* is defined as an unordered pair of clone regions which have similar normalized instructions. A *clone cluster* is a group of clone pairs. The exact clone detector identifies clones among the regions by comparing their instruction mnemonics. Two regions are considered an exact clone if and only if all the normalized instructions in the two regions are identical. A naïve approach to identify exact clones would be to compare every region pair. Yet, this approach is too computationally expensive with a complexity of $O(n^2)$, where n is the total number of regions. Thus, a hashing approach is used. Specifically, two regions are considered an exact clone

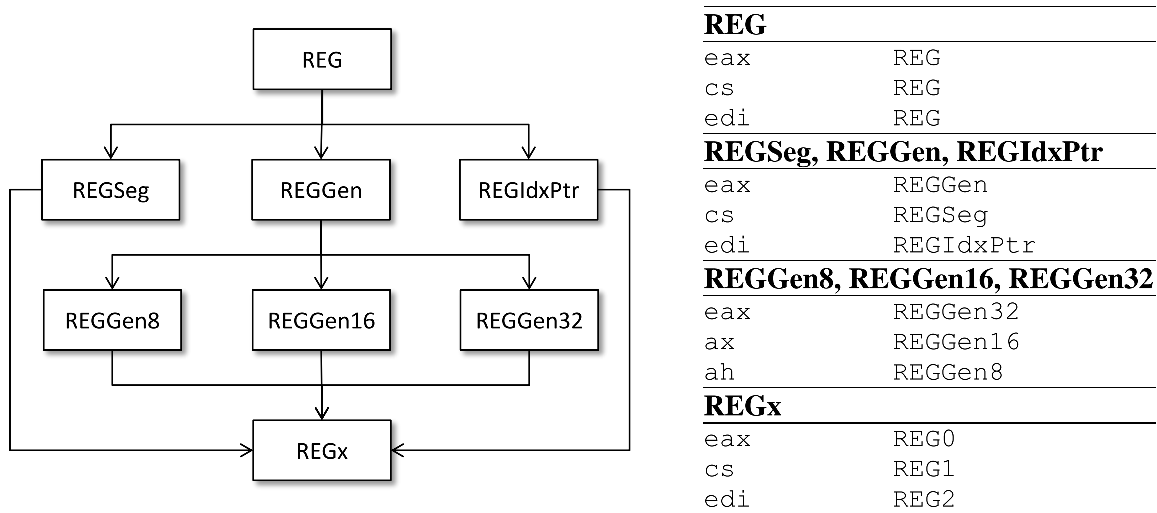
if they share the same hash value. The exact clone detector is an improvement over the work of Schulman [19].

5. **Inexact clone detector:** This step extracts features for each region and forms a feature vector, denoted by *v*, for each region. Two regions r_x and r_y are considered an inexact clone if the similarity between their feature vectors, denoted by $sim(v_x, v_y)$, is within a user-specified minimum similarity threshold *minS*. Section 4.3 explains this process in details. The resulting set of identified clone clusters might contain many overlapping regions that are meaningless for analysis purposes. This happens when the step size *s* is smaller than the window size *w*. A post-processor identifies and removes these trivial overlapping instruction sequences.
6. **Duplicate clone merger:** The inexact clone detector might misclassify two consecutive regions as a clone. The duplicate clone merger removes clones that are just highly overlapping consecutive regions. This also happens when the step size *s* is smaller than the windows size *w*.
7. **Maximal clone merger:** Since the clone detection process operates on regions, the maximum size of the identified clones will correspond to the region size. This prevents the identification of cloned fragments spread over consecutive cloned regions. As it is more useful to identify a large clone than several smaller ones, the seventh step merges consecutive cloned regions into a larger clone.
8. **Token indexer:** Separately from the aforementioned clone detection process, this step parses the assembly files to create indexes for constants, strings, and imported function names. The goal is to facilitate the direct access to these tokens during code clone search.
9. **Visualizer:** A graphical user interface was also implemented to allow users input the required parameters for code clone detection, specify target code fragments or tokens, and display the matched clone fragments or tokens from the assembly files.

For more details about the regionizer, exact clone detector, duplicate clone merger, and maximal clone merger processes, refer to [18]. In the remaining of this section, the improvements and extensions made in this research compared to the original work of Saebjornsen et al. [18] are described.

4.2 Normalizer

In assembly code, an instruction typically consists of a mnemonic (e.g., `mov`) and an operands list. Possible operands can be a register (e.g., `eax`), a constant (e.g., `0x30004040`), or a memory address (e.g., `[0x4000349e]`). As two or more code regions can be similar except for differences in the instructions operands used, these need to be normalized in order to take into account these variations. Different works in the literature were investigated and extensive experiments were performed on assembly code samples. These revealed that different normalization techniques can result in significantly different clones. Therefore, to add flexibility to the clone detection and search process, the following normalization scheme was implemented. A constant can be normalized to `VAL` or `VALx`, where `x` is an index number. Similarly, a memory address can be normalized to `MEM` or `MEMx`. Registers can be normalized according to the hierarchy shown in Figure 3. This figure also illustrates how the `EAX`, `CS`, and `EDI` registers would be mapped according to the different normalization levels.



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

Figure 3: Normalization hierarchy for registers and mapping examples

Using the more abstract normalization level, Figure 4 illustrates how some sample assembly code instructions would be normalized.

Table 1: Normalized assembly code instructions

Assembly Code	Normalized Assembly Code
`mov edi, edi`	`mov REG, REG`
`push ebp`	`push REG`
`push ebp, esp`	`push REG, REG`
`mov eax, dword ptr [epb+8]`	`mov REG, MEM`

4.3 Inexact Clone Detector

In [18], Saebjornsen et al. proposed an inexact clone detector to identify clone pairs that are not exactly identical. In general, their approach consists of first extracting a set of features from each region and then searching for other code regions with the same or similar feature set. Specifically, a feature vector is constructed based on the following five types of features from each region [18]:

- *M*, representing the mnemonic of the instruction
- *OPTYPE*, representing the type of each operand in an instruction
- *M* × *OPTYPE*, representing the combination of the mnemonic and the type of the first operand, when one is present
- *OPTYPE* × *OPTYPE*, representing the types of the first and second operands, in that order, of an instruction with at least two operands
- *OPTYPE* × N_k, representing each normalized operand with an index under a chosen limit k

Using the same set of features, a new approach which can efficiently identify inexact clone pairs is proposed.

Its algorithm can be described in the following four steps:

1. **Compute median vector:** The median of each feature for all regions is computed. The resulting vector is called the *median vector*. Intuitively, a feature having a median equal to zero implies that the majority of regions do not contain this feature. It should thus be removed, as it cannot be used to differentiate regions.
2. **Compute binary vectors:** A *binary vector* is computed for each region by comparing the value of a feature vector with the corresponding value in the median vector. If the feature value is larger than the corresponding median, then 1 is inserted into the binary vector. Otherwise, 0 is inserted. For a region with feature values *<0, 2, 1, 4, 1>*, its binary vector would be *<0, 0, 0, 1, 0>* with respect to the median vector *<1, 5, 2, 3, 3>*.
3. **Hash binary vectors:** For each binary vector, a hash key of every *k* consecutive features is iteratively computed, where *k* is a user-specified parameter. The regions having the same hash key are put into the same bucket of a hash table. For example, Table 2 shows that regions 6, 7, 33, and 76 are hashed into the same bucket with respect to the first five consecutive features. The number of hash tables is bounded by the size of the binary vectors, i.e., the number of features having non-zero medians.

Table 2: Hash table for inexact clone detection

Key	Values (Region No.)
0	8, 9, 22, 156
1	6, 7, 33, 76
2	0, 56, 87, 12
…	…
31	53, 21, 1, 9

4. **Construct clone pairs:** Intuitively, regions that frequently appear together in the same buckets of different hash tables are similar. They should therefore form a clone pair. The co-occurrence of regions can be computed by simply scanning the hash tables and keeping track of the co-occurrence counts using a score table such as Table 3. For example, for hash key 0 in Table 2, the scores of {8, 9}, {8, 22}, {8, 156}, {9, 22}, {9, 156}, and {22, 156} are incremented by 1. Similarly, for hash key 31, the scores of {53, 21}, {53, 1}, {53, 9}, {21, 1}, {21, 9}, and {1, 9} are also incremented by 1. The pairs of regions having a score above the user-specified threshold *minS* are considered as clone pairs.

Table 3: Score table for inexact clone detection

Region No.	0	1	2	3	4	…	N
0	-	3	1	1	1	…	12
1	-	-	4	2	8	…	4
2	-	-	-	6	6	…	0
3	-	-	-	-	5	…	0
4	-	-	-	-	-	...	1
…	…	…	…	…	…	…	…
N	-	-	-	-	-	…	-

5.0 EMPIRICAL STUDY

The objective of the empirical study was to evaluate the proposed assembly code clone search approach in terms of precision, efficiency, and scalability. Experiments were conducted using three different sets of binary files. The first set contains two well-known malware: Zeus and Blaster. Zeus [5] is a Trojan horse that attempts to steal confidential information from a compromised computer. Blaster [1] is a worm that propagates by exploiting a buffer overflow vulnerability in the Microsoft Windows Remote Procedure Call (RPC) interface. The second set is a collection of 70 malware obtained from the National Cyber-Forensics and Training Alliance (NCFTA) Canada. The third and final set is an assortment of 18 open source Dynamic Link Libraries (DLLs). The experiments were performed on an Intel Xeon X5460 3.16 GHz Quad-Core processor-based server with 48GB of RAM running Windows Server 2003.

5.1 Accuracy

To evaluate the accuracy of the proposed approach, 20 code fragments were first selected from the 18 disassembled DLLs and clones of these code fragments were manually identified in the assembly files. Then, the manually identified clones were compared with the results generated by the implemented code clone search approach to compute the following three measures:

- $Precision(Solution, Result) = \frac{n_{ij}}{|Result|}$
- $Recall(Solution, Result) = \frac{n_{ij}}{|Solution|}$
- $F(Solution, Result) = \frac{2 \times Recall(Solution,Result) \times Precision(Solution)}{Recall(Solution,Result)+Precision(Solution,Result)}$

where *Solution* is the set of manually identified code clones, *Result* is the set of code fragments in a 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 search *Result* with respect to the *Solution* by the harmonic mean of *Recall* and *Precision*. As the goal is to evaluate the quality of the search results with respect to a manually identified solution, it is infeasible to perform this experiment on a large collection of assembly files.

Figure 4 shows the resulting precision, recall, and F-score measures for two different minimum similarity thresholds *minS* (0.5 and 0.8) using a step size $s = 1$ and a maximum number of features $k = 40$. Recall, precision, and F-score are consistently above 80% for different window sizes, suggesting that the clone detection method is accurate.

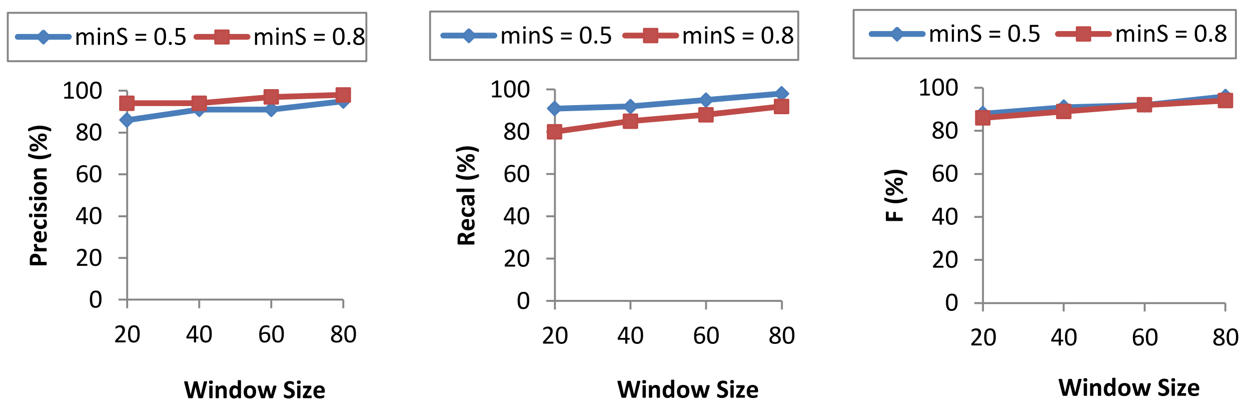


Figure 4: Accuracy for *s* = 1 and *k* = 40 (open source DLLs)

To evaluate the precision of the proposed clone search method using Zeus and Blaster, the first 10 regions of each malware were selected as target code fragments. Clones of each selected region were then searched in the rest of the assembly code. Each identified clone was then manually reviewed to determine whether it was a valid clone or not. Using a step size $s = 1$, a maximum number of features $k = 40$, a minimum similarity threshold $minS = 0.8$, and a window size w ranging from 20 to 80, the precision was consistently above 96%. The approach was also applied on the collection of 70 malware to evaluate the number of both exact and inexact clones detected. Table 4 shows the numbers for various window sizes.

Table 4: Number of exact and inexact clones detected (malware assortment)

Window Size	Exact Clones	Inexact Clones
20	18,010	26,6335
40	17,225	27,2008
60	17,162	27,4346
80	16,971	75,9953

5.2 Efficiency

Figure 5 depicts the runtime in seconds for the exact, inexact, and for both clone detection using the following parameters: $s = 1$, $k = 40$, and $minS = 0.80$. The sample set was the Zeus and Blaster malware, and various window sizes ranging from 20 to 80 were used. The clone detection process took between 23 and 30 seconds, indicating that its efficiency is not sensitive to the window size.

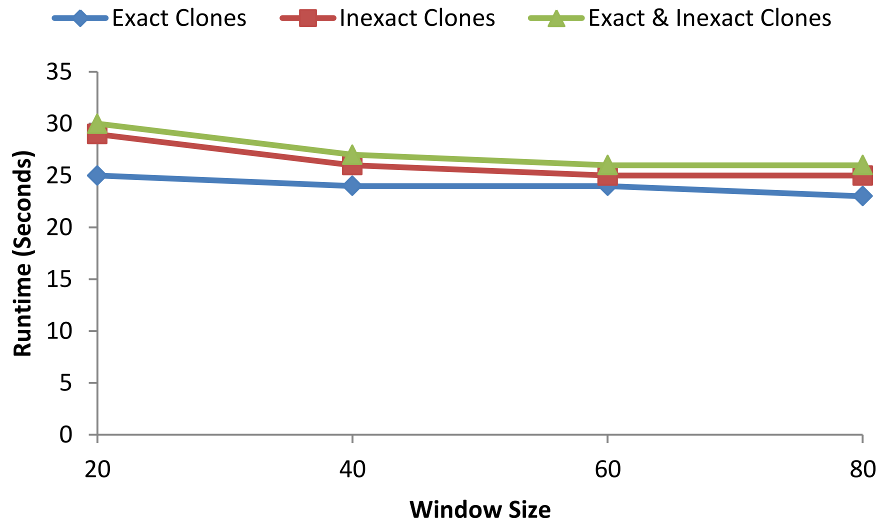


Figure 5: Runtime vs. window size (Zeus and Blaster malware)

5.3 Scalability

Figure 6 illustrates the runtime in seconds for the different steps of the process using 10 to 70 malware and the following parameters: $s = 1$, $k = 40$, and $minS = 0.80$. The first step reads and processes the data. The second and third step respectively detects exact and inexact clones. Step 4 merges the clones and finally, the results are saved into an XML file. The total processing time ranges from 8 to 258 seconds.

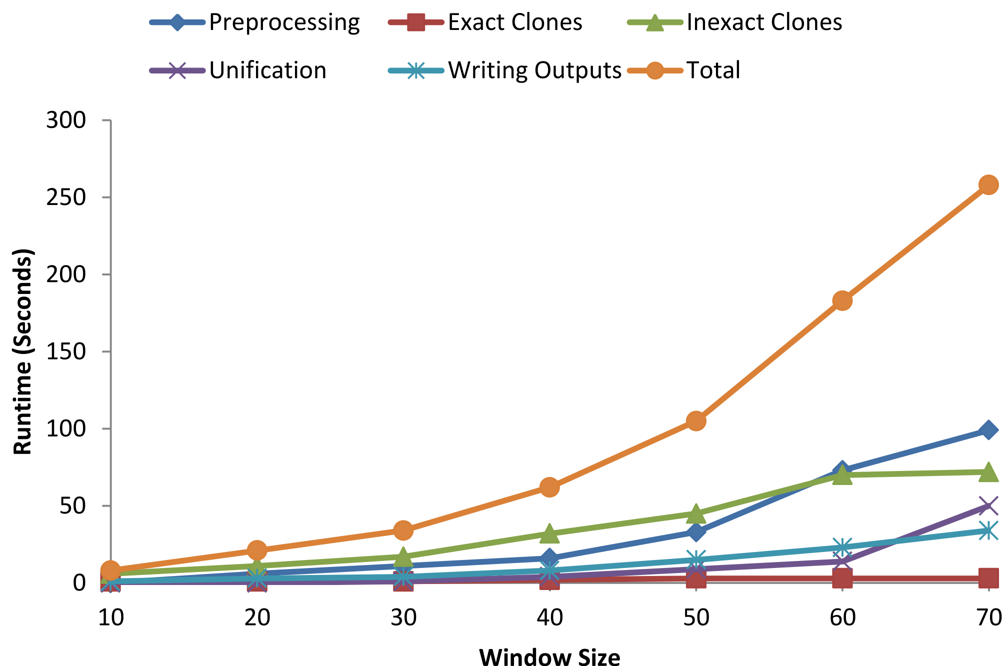


Figure 6: Scalability (malware collection)

6.0 CONCLUSION AND FUTURE WORK

In this article, the prototype of a clone search system for malware analysis was implemented. It expands on the work of Saebjornsen et al. [18] through several improvements and extensions. First, a flexible normalization scheme was implemented. Second, a new inexact clone detection method was developed. Third, a search capability on constants, strings, and imported function names was added. Finally, a graphical user interface was implemented to browse and visualize the identified clones. The performance of the clone search system was evaluated in terms of accuracy, efficiency, and scalability. Experimental results suggest that the implemented clone search algorithm is effective at identifying both exact and inexact clones in assembly code. The current prototype implementation, like most of the works in the literature, supports the identification of syntactic clones (Type I, II, and III). The identification of semantic clones (Type IV) remains a challenging research problem for both source and assembly code. Future work will consist of investigating other approaches for identifying semantic clones in assembly code and conducting additional case studies to validate them.

7.0 REFERENCES

[1] M. Bailey, et al., “The Blaster Worm: Then and Now,” *IEEE Security and Privacy*, vol. 3, no. 4, Jul. 2005, pp. 26-31.

[2] B.S. Baker, “On Finding Duplication and Near-Duplication in Large Software Systems,” *Proc. of the 2nd Working Conf. on Reverse Eng. (WCRE ‘95)*, Toronto, Ont., Jul. 1995, pp. 86-95.

[3] H.A. Basit, et al., "Efficient Token Based Clone Detection with Flexible Tokenization," *Proc. of the 6th Joint Meeting of the European Software Eng. Conf. and the ACM SIGSOFT Symp. on the Foundations of Software Eng*., Dubrovnik, Croatia, Sept. 2007, pp. 513-516.

[4] I.D. Baxter et al., "Clone Detection Using Abstract Syntax Trees," *Proc. of the Int'l Conf. on Software Maintenance (ICSM '98)*, Bethesda, Md., Nov. 1998, pp. 368-377.

[5] H. Bin, et al., "On the Analysis of the Zeus Botnet Crimeware Toolkit," *Proc. of the 8th Ann. Conf. on Privacy, Security and Trust (PST 2010)*, Ottawa, Ont., Aug. 2010, pp. 31-38.

[6] W.S. Evans, C.W. Fraser, and F. Ma, "Clone Detection via Structural Abstraction," *Proc. of the 14th Working Conf. on Reverse Eng. (WCRE '07)*, Vancouver, B.C., Oct. 2007, pp. 150-159.

[7] Hex-Rays, "IDA: About," Aug. 2012; http://www.hex-rays.com/products/ida/index.shtml.

[8] B. Hummel, et al., "Index-Based Code Clone Detection: Incremental, Distributed, Scalable," *Proc. of the IEEE Int'l Conf. on Software Maintenance (ICSM '10)*, Timisoara, Romania, Sept. 2010, pp. 1-9.

[9] J. Ji, et al., "Source Code Similarity Detection Using Adaptive Local Alignment of Keywords," *Proc. of the 8th Int'l Conf. on Parallel and Distributed Computing, Applications and Technologies (PDCAT '07)*, Adelaide, Australia, Dec. 2007, pp. 179-180.

[10] J.H. Johnson, "Identifying Redundancy in Source Code Using Fingerprints," *Proc. of the 1993 Conf. of the Centre for Advanced Studies on Collaborative Research: Software Eng.*, Toronto, Ont., Sept. 1993, pp. 171-183.

[11] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code, *IEEE Trans. on Software Eng.*, vol. 28, no. 7, Jul. 2002, pp. 654-670.

[12] R. Komondoor and S. Horwitz, "Using Slicing to Identify Duplication in Source Code," *Proc. of the 8th Int'l Symp. on Static Analysis (SAS '01)*, Paris, France, Jul. 2001, pp. 40-56.

[13] K.A. Kontogiannis, et al., "Pattern Matching for Clone and Concept Detection," *J. of Automated Software Engineering*, vol. 3, no. 1-2, Jun. 1996, pp. 77-108.

[14] J. Krinke, "Identifying Similar Code with Program Dependence Graphs," *Proc. of the 8th Working Conf. on Reverse Eng. (WCRE '01)*, Stuttgart, Germany, Oct. 2001, pp. 301-309.

[15] C. Liu, et al., "GPLAG: Detection of Software Plagiarism by Program Dependence Graph Analysis," *Proc. of the 12th ACM SIGKDD Int'l Conf. on Knowledge Discovery and Data Mining (KDD '06)*, Philadelphia, Pa., Aug. 2006, pp. 872-881.

[16] A. Marcus and J.I. Maletic, "Identification of High-level Concept Clones in Source Code," *Proc. of the 16th IEEE Int'l Conf. on Automated Software Eng. (ASE '01)*, Coronado, Calif., Nov. 2001, pp. 107-114.

[17] J. Mayrand, C. Leblanc, and E. Merlo, “Experiment on the Automatic Detection of Function Clones in a Software System Using Metrics,” *Proc. of the 1996 Int’l Conf. on Software Maintenance (ICSM ’96)*, Monterey, Calif., Nov. 1996, pp. 244-253.

[18] A. Saebjornsen, et al., “Detecting Code Clones in Binary Executables,” *Proc. of the 18th Int’l Symp. on Software Testing and Analysis (ISSTA '09)*, Chicago, Ill., Jul. 2009, pp. 117-128.

[19] A. Schulman, “Finding Binary Clones with Opstrings & Function Digests,” *Dr. Dobb’s Journal*, Jul. 2005 (Part I), Aug. 2005 (Part II), and Sept. 2005 (Part III).

[20] C.K. Roy and J.R. Cordy, *A Survey on Software Clone Detection Research*, tech. report 2007-541, School of Computing, Queen's Univ., Kingston, Ont., 2007.

[21] C.K. Roy, J.R. Cordy, and R. Koschke, “Comparison and Evaluation of code Clone Detection Techniques and Tools: A Qualitative Approach,” *Science of Computer Programming*, vol. 74, no. 7, May 2009, pp. 470-495.

[22] V. Wahler, et al., “Clone Detection in Source Code by Frequent Itemset Techniques,” *Proc. of the 4th IEEE Int’l Workshop on Source Code Analysis and Manipulation (SCAM ‘04)*, Chicago, Ill., Sept. 2004, pp. 128-135.