SERBIAN JOURNAL OF ELECTRICAL ENGINEERING
Vol. 12, No. 2, June 2015, 253-262

UDC: 004.4`22:[004.415.3+004.424.46 **DOI: 10.2298/SJEE1502253B**

Estimation of Similarity between Functions Extracted from x86 Executable Files

Katarina Berta[1], Saša Stojanović[1], Miloš Cvetanović[1], Zaharije Radivojević[1]

Abstract: Comparison of functions is required in various domains of software engineering. In most domains, comparison is done using source code, but in some domains, such as license violation or malware analysis, only binary code is available. The goal of this paper is to evaluate whether the existing solution meant for ARM architecture can be applied to x86 architecture. The existing solution encompasses multiple approaches, but for the purpose of this paper three representative approaches are implemented; two are based on machine learning, and the third does not require previous knowledge. Results show that the best recalls obtained for the first ten positions on both architectures are comparable and do not differ significantly. The results confirm that adaptation of all approaches of the existing solution is not only possible but also promising and represent adequate basis for future research.

Keywords: Code analysis, Binary code, Software clones, Similarity assessment.

1 Introduction

Function similarity estimation is a part of the more general field of code similarity analysis. Studying code similarity in most domains is based on source code analysis, mostly with the intention to identify software clones [1]. In cases when only a binary code is available, analysis primarily depends on the application domain. In the malware detection domain, binary code analysis is used for recognizing functions similar to the known functions with malicious behavior [2]. Similarly, in the domain of vulnerability analysis, multiple versions of the same binary code are compared with an aim to identify changes that represent potential security threats [3]. The basis of this paper is the solution intended for the domain of license violation [4].

In general, estimation of similarity between functions from executable files can be defined as the following problem: is it possible to determine a measure that achieves maximal value when two compared functions found in two different executable files, more precisely their assemblies, originate from the

[1]School of Electrical Engineering, University of Belgrade, Bul. kralja Aleksandra 73, 11120 Belgrade, Serbia; E-mails: katarinaberta@gmail.com; stojsasa@etf.bg.ac.rs; cmilos@etf.bg.ac.rs; zaki@etf.bg.ac.rs

same function in the source code, although they have not been compiled using the same compiler? The existing solution solves this problem by using software metrics extracted from binaries compiled for ARM architecture. Software metrics are used for calculating partial similarities that are combined using different approaches for obtaining total similarity. The goal of this paper is to evaluate applicability of the existing solution on binaries compiled for x86 architecture. For this evaluation, three approaches are implemented, two based on machine learning (weighted sum and Naïve Bayes) and the third that does not require previous knowledge (harmonic mean). Each of the implemented approaches combines four different software metrics to obtain the best result. The software metrics considered in this paper represent the subset of the metrics proposed by the existing solution. They are number of jumps in function, number of function calls inside the function body, function length, and list with number of appearance of different assembly instructions inside the function body.

The rest of this paper is organized as follows. Section 2 presents theoretical background for this paper. Section 3 describes the main steps of the algorithm used for estimating function similarity. Section 4 sheds more light on amounts of differences that compilers introduce. Section 5 describes conducted experiments and shows obtained results, while Section 6 concludes the paper.

2 Code Analysis and Code Similarity

The code analysis can be conducted in a dynamic or a static manner. Dynamic code analysis is conducted on a code that is executed with an example input data. Therefore, various input data should be used to cover most of the execution paths and make the analysis as comprehensive as possible. Static code analysis is a process of analysing code without executing the code. Although dynamic code analysis due to execution of the code provides information that might be highly useful for the purpose of the analysis, static code analysis does not require knowledge related to preparation of input data nor additional time for multiple executions of the code [5]. An important usage of static code analysis, the one concerning this paper, is code similarity evaluation. For the purpose of the analysis, either source code or binary code can be used as an input.

Using source code as an input is widely present in detection of plagiarism among student projects [6] and detection of duplicated code in large software systems [7]. Commercial solutions are available for code clone detection, even as a part of integrated development environments [8]. Analysis can be split in three steps: source code parsing, generating the internal representation, and analysis of the internal representation. Complexity of each step varies depending on robustness of the analyzer [9]. Source code parsing generates

input for the process of translating code to some other representation, more convenient for analysis. Internal representation can be as simple as a collection of tokens [10], or as complex as a program dependence graph [11]. Analysis of internal representation is the most challenging part of the process.

Binary code is used as an input for plagiarism detection in commercial code, malicious software analysis, and other similar cases when source code is not available. The first step of binary code analysis, disassembling, produces assembly listings which are analyzed further. The rest of the analysis steps can, in general, be described in the same way as for the source code analysis. However, the algorithms and the implementation of every step are different because binary code lacks much of the data present in source code, in the first place symbol names.

3 Tool for Estimating Function Similarity

For the purpose of this research, a prototype software tool for function similarity analysis based on executable files is developed. The tool implements three selected approaches for x86 executables, according to research that was intended for ARM architecture. The tool is implemented in Java, and therefore is a cross-platform portable solution. As input, the tool takes two files, or lists of files with assembly instructions. The input files are analyzed through three steps: input file parsing, generating the internal representation, and analysis of the internal representation, as described in the following subsections.

The parser reads assembler instructions from input files. Currently, the parser supports only input file format that is produced as output of the IDA disassembler [12]. In addition to reading assembly input files, the tool implements a special parser for .map files which extracts function names and addresses and matches them with addresses from assembly files. This feature is used only during evaluation of the tool's success in estimation of function similarity.

After parsing, all read instructions along with some additional data are stored in internal representation. The basic unit of internal representation is a structure containing function statistics. The structure stores the following metrics: length of a function measured as number of assembler instructions, number of all branch instructions, number of all CALL instructions, and a list of pairs containing an instruction name and the number of times the instruction appears in the function body. Currently, the structures describing functions are stored in a linked list. However, during parsing there is a support for making a call graphs that would enable not only estimating similarity between functions, but also comparison of whole programs.

The tool uses internal representation to estimate similarity between one function from the first input file (searched function) and all functions from the

second input file. The functions from the second file are ranked according to the estimated similarity, and the tool returns a specified number of most similar functions.

Similarity assessment is implemented in two levels. The first level consists of matchers. Each matcher compares only one software metric of functions, and returns a value between 0 and 1. Value 1 states that the compared functions are the same according to the metric used. Implemented matchers compare frequency or number of instructions in functions, jumps, calls, number of occurrences of same instructions, and ascending list of number of occurrences of instructions.

The second level of comparison uses different formulas for combining matcher results, giving them more or less weight and significance. Each formula can be used independently and gives a final evaluation of function similarity. Currently, three formulas are implemented: harmonic mean, weighted sum, and Naïve Bayes. Evaluation of implemented matchers and formulas is presented in Section 5.

4 Influence of Compilers

Estimating similarity between functions extracted from executable files is highly influenced by the compiler that is used. Different compilers use various algorithms for register allocation, calling conventions, and inline optimizations. The amount of differences introduced by compilers is reflected through characteristics of generated code such as code size and number of functions. To illustrate the amount of differences, **Table 1** gives some characteristics of executable files that originate from the same source code, but were compiled using different compilers. The table shows function-related characteristics for different compilers based on parsing corresponding disassembled executable files.

The source code used for collecting characteristics originates from Bayes, which is one of the programs from STAMP benchmark (Stanford Transactional Applications for Multi-Processing), written in C/C++ [13]. The source code is compiled with the following compilers: Borland C++ Builder 2010, Embarcadero RAD Studio 2010, Dev C++ Bloodshed Software, Open Watcom C/C++ 1.8, Microsoft Visual C++ 6.0 Enterprise Edition (VS6), Visual C++ 2008 Express Edition, and Intel C++ Compiler Professional Edition. The set of compilers is selected in such a way to cover most commonly used C/C++ compilers for the Windows operating system. Moreover, all characteristics are collected for both modes of compilation, debug and release.

The first column represents the number of defined functions in the executable, meaning the total number of function bodies in the executable file. As a reference, by parsing Bayes program source code, 90 functions are

detected. All executable files contain more than 90 functions because of statically linked libraries. The release mode produces less or the same number of functions than debug for the same compiler. Comparing numbers between different compilers and compilation modes, the large differences can be noticed (e.g. 111 for Intel compiler in Release mode vs. 650 for Visual Studio 6 compiler in Debug mode).

The second column shows the cumulative number of called "undefined" functions. The term "undefined" accounts for DLL functions and other functions that appear as a target of CALL instructions and whose definition could not be found inside the source file. Large variations between compilers are present, meaning that different compilers use various libraries to achieve the same functionality.

Table 1
Characteristics of disassembled Bayes executable files generated using different compilers.

Compilers	Mode	Defined functions	Called undefined functions	Called function	Defined called functions	Defined not called functions	Unrecognized part of file [%]
Borland	D	154	54	194	140	14	1.85
	R	154	52	192	140	14	2.12
Dev	D	229	59	238	179	50	32.08
	R	229	59	238	179	50	32.08
Watcom	D	414	66	385	319	95	24.63
	R	346	66	346	280	66	26.22
VS6	D	650	241	557	316	334	1.29
	R	480	220	441	221	259	2.36
VS2008	D	368	304	411	107	261	0.76
	R	338	65	280	215	123	0.03
Intel	D	357	243	387	144	213	0.25
	R	111	67	156	89	22	1.60

The third column shows the number of all functions called along the assembler code, while the fourth and fifth columns show the number of defined functions that are called, and that are not called, respectively. Differences between compilers regarding the fourth and fifth columns are big and range from two to three times. Moreover, the differences present in the first two columns are even bigger and range up to six times.

The last column shows the percent of the executable file that is not recognized as a part of any function. For the Dev compiler, the unrecognized part is 32.08%, leading to a conclusion that effectiveness of similarity estimation highly depends on the quality of disassembling because cases could exist in which entire functions are not recognized.

5 Approaches and Evaluation

The main goal of this paper is to evaluate whether the existing approaches proposed for ARM architecture can be applied with similar success rate for x86 architecture. For the purpose of the evaluation, the parser for x86 is implemented, software metrics and appropriate metrics are adapted, and the following three formulas are implemented and tested: harmonic mean, weighted sum, and Naïve Bayes. In the rest of this section, the experiment setup is described, followed by a description of the approaches and results obtained.

5.1 Experiment setup

Evaluation of the approaches is accomplished using seven STAMP programs that were compiled with compilers mentioned in Section 4, in both debug and release mode. Executable files were then disassembled using IDA. Assembly listings obtained by disassembling are parsed together with corresponding map files to associate names to functions. The names of the functions are used in evaluation only to check the success rate of the approaches, because it was the only way to know if two functions really originate from same source code function.

Only functions in disassembled code whose names exist in the appropriate map file were used in the evaluation as a searched function. The functions were used in the evaluation in two phases: training and testing. For the first phase, training, a dataset of 2762 pairs of functions was chosen randomly. For the second phase, testing, a dataset of 1372 tests was created. Each test consists of one randomly chosen searched function and one randomly chosen binary file containing the function. The function chosen for a test was compared with each function from the file chosen for the same test. Results of the comparison are estimations of similarity which are used to rank the functions from the file.

Comparison of the functions and estimation of the similarity is conducted using the prototype tool described in Section 3. Statistics produced by the tool were made by counting true positive matches expressed as a recall on the first position and by the third, fifth, and tenth position. The true positive is recognized by comparing the names associated to the compared functions.

The evaluation compared recall for all three implemented formulas. Each formula tried to combine four matchers that are result of comparison based on appropriate metrics: function length matcher (LN), branch count matcher (BC),

call count matcher (CC), and instruction count matcher (IC). The first three are self-explanatory, and they are calculated as a ratio between smaller and bigger metric's value of two compared functions. The fourth compares lists containing numbers of occurrences for each instruction by calculating average value of ratios of smaller and larger number of occurrences for each instruction.

5.2 Setup of approaches

Harmonic mean is one of the Pythagorean means, along with arithmetic and geometric mean, and its value is not greater than any of these two [14]. Harmonic mean is the first evaluated formula and was chosen because of its property to reduce the effect of outliers. For example, when comparing functions that have zero jumps, the appropriate matcher would give one, which would influence the final result, even though the case might be that two completely different short functions are compared. Moreover, the harmonic mean is the only formula among those used that does not require previous knowledge.

Weighted sum is the second evaluated formula and is calculated as a sum of matchers multiplied by the weight factor associated with each matcher. The formula is chosen because it is one of the natural choices for problems of combining more factors. The most challenging part of using the weighted sum is deciding on factor values; in the evaluation, it was done using curve fitting on the training dataset. Fitting was targeted to minimize the sum of error squares, where error was either equal to the weighted sum of matchers, for pairs of functions that are not matches, or one minus weighted sum for actual matches. Obtained weights are: -0.61 for LN, 1.33 for IC, 0.35 for BC, and -0.04 for CC. The final version of formula implies that IC matcher is far superior to the other matchers.

The third evaluated formula, Naïve Bayes classifier, is a simple probabilistic classifier based on applying Bayes' theorem with a strong independence assumption. The independence assumption is not realistic, but in practice this classifier gives good results, even for a small training dataset, and is often used for benchmarking [15]. As for every machine learning algorithm, training is the first step of applying the Naïve Bayes classifier. The training dataset is used to calculate means and standard deviations for each matcher, for two classes (true positive matches and false positive matches), and for prior probability of both classes, a value of 0.5 is used. The setup of the Naïve Bayes formula is given in **Table 2**, where rows represent results for the appropriate matcher. The similarity estimation between two functions is obtained by calculating a conditional probability of the functions being a true or false positive match, using Bayes' theorem [16].

Table 2
Means and standard deviations for Naïve Bayes.

Matcher	pair		not a pair	
	Mean	std.dev	mean	std.dev
LN	0.89	0.20	0.72	0.33
BC	0.93	0.18	0.71	0.35
CC	0.90	0.23	0.68	0.40
IC	0.76	0.25	0.35	0.16

5.3 Results

Average recall on the first and by the third, fifth, and tenth position for three formulas are presented in **Table 3**. Comparison of results of all three formulas shows similar performances. Slightly better results are achieved by Naive Bayes and weighted sum than by harmonic mean. It is interesting to notice that Naive Bayes and weighted sum are trained using previous knowledge, while harmonic mean does not use previous knowledge. The best results are obtained by using weighted sum with weights obtained using the training dataset. The weighted sum matches more than 50% of functions already in the first three positions. The recall on the first position of the approach highly depends on the compiler pair used for compilation of compared codes. If the compared codes are compiled with same compiler, recall on the first position increases up to 60%, while on the first ten positions it increases up to 86%.

Table 3
Recall in the first ten positions for evaluated approaches on x86 architecture.

Formula	1.	3.	5.	10.
Harmonic mean	34.86	47.52	52.33	58.50
Weighted sum	37.62	52.14	56.94	63.03
Naïve Bayes	36.14	48.77	52.91	58.78

For the purpose of the evaluation, the results on x86 are compared with results obtained on ARM using tests from the same benchmark programs with the same set of matchers and formulas. The best results obtained for the first position on x86 architectures are better than results on ARM architecture almost 26% (37.62 for x86 vs. 29.91 for ARM). However, for increasing number of observed positions, the difference decreases. Even more, observing the first ten positions results in ARM architecture becoming slightly better (63.03 for x86 vs. 65.12 for ARM).

6 Conclusion

This paper evaluated applicability of the existing solution for comparison of ARM binary procedures on code compiled for x86 architecture. For the purpose of the evaluation, three approaches were implemented. Two are based on machine learning (weighted sum and Naïve Bayes), and the third does not require previous knowledge (harmonic mean). Each of the implemented approaches combines four different software metrics that are a subset of the metrics proposed by the existing solution. The evaluation was conducted using seven STAMP programs that were compiled with six different compilers using the debug and release mode.

Results for the x86 architecture show that the best recall was achieved by the approach based on the weighted sum. It ranges from 37% to 63% when observing the first and first ten positions, respectively. Comparing the results between x86 and ARM architectures reveals that the best recalls obtained for the first ten positions on both architectures are comparable. An interesting topic for further research would be to conduct a new evaluation that would adapt all approaches originally intended for ARM architecture and apply them on the x86 architecture.

7 Acknowledgements

Work on this project was cofunded by the Ministry of Education, Science, and Technological Development of the Republic of Serbia (III44009 and TR32047). The authors gratefully acknowledge the support.

8 References

[1] D. Rattan, R. Bhatia, M. Singh: Software Clone Detection: A Systematic Review, Information and Software Technology, Vol. 55, No. 7, July 2013, pp. 1165 – 1199.

[2] I. Santos, F. Brezo, J. Nieves, Y. Penya, B. Sanz, C. Laorden, P.G. Bringas: Idea: Opcode-Sequence-based Malware Detection, Engineering Secure Software and Systems, Vol. 5965, Jan. 2010, pp. 35 – 43.

[3] T. Dullien, R. Rolles: Graph-based Comparison of Executable Objects, Symposium on the Safety of Information and Communication Technologies, Rennes, France, 01-03 June 2005, pp. 1 – 13.

[4] S. Stojanović, Z. Radivojević, M. Cvetanović: Approach for Estimating Similarity Between Procedures in Differently Compiled Binaries, Information and Software Technology, Vol. 58, Feb. 2015, pp. 259 – 271.

[5] P. Louridas: Static Code Analysis, IEEE Software, Vol. 23, No. 4, July/Aug. 2006, pp. 58 – 61.

[6] T. Lancaster, C. Fintan: A Comparison of Source Code Plagiarism Detection Engines, Computer Science Education, Vol. 14, No. 2, 2004, pp. 101 – 112.

[7] S. Livieri, Y. Higo, M. Matushita, K. Inoue: Very-large Scale Code Clone Analysis and Visualization of Open Source Programs using Distributed CCFinder: D-CCFinder, 29th

International Conference on Software Engineering, Minneapolis, MN, USA, 20-26 May 2007, pp. 106 – 115.

[8] http://msdn.microsoft.com/en-us/library/hh205279.aspx.

[9] D. Binkley: Source Code Analysis: A Road Map, IEEE Future of Software Engineering, Minneapolis, MN, USA, 23-25 May 2007, pp. 104 – 119.

[10] Y. Yuan, G. Yao: Boreas: An Accurate and Scalable Token-based Approach to Code Clone Detection, 27th IEEE/ACM International Conference on Automated Software Engineering, Essen, Germany, 03-07 Sept. 2012, pp. 286 – 289.

[11] M. Gabel, J. Lingxiao, S. Zhendong: Scalable Detection of Semantic Clones, 30th International Conference on Software Engineering, Leipzig, Germany, 10-18 May 2008, pp. 321 – 330.

[12] https://www.hex-rays.com/products/ida/.

[13] C.C. Minh, J.W. Chung, C. Kozyrakis, K. Olukotun: STAMP: Stanford Transactional Applications for Multi-Processing, IEEE International Symposium on Workload Characterization, Seattle, USA, 14-16 Sept. 2008, pp. 35 – 46.

[14] D.W. Mitchell: More on Spreads and Non-Arithmetic Means, The Mathematical Gazette, Vol. 88, No. 511, March 2004, pp. 142 – 144.

[15] L.I. Kuncheva: On the Optimality of Naive Bayes with Dependent Binary Features, Pattern Recognition Letters, Vol. 27, No. 7, May 2006, pp. 830 – 837.

[16] B. Efron: Bayes' Theorem in the 21st Century, Science, Vol. 340, No. 6137, June 2013, pp. 1177 – 1178.