Comparing Assembler Procedures by Analyzing Sequences of Opcodes

Nikola Pejić, Miloš Cvetanović, and Zaharije Radivojević

***Abstract* — Static analysis of executables for the purpose of comparing them can be made more difficult if the binaries are created using different compilers. In order to compensate for the noise introduced by the compilers, the arguments of the instructions are usually discarded as having a low signal-to-noise ratio. As compiler can often apply instruction reordering, some approaches only compare statistical information about the instructions, or compare their subsequences in order to measure their similarity.**

This paper presents an approach for estimating the similarity of procedures given in assembler form (disassembled binaries) by analyzing their sequences of opcodes. The approach first encodes the opcodes into integer values by mapping opcodes that represent similar actions into the same values, and then calculates a relative Levenshtein distance between the two sequences of integers. The proposed approach is evaluated and compared with some existing approaches, where it showed to have on average around 6% higher recall than the second-best approach.

***Keywords* — assembler code analysis, sequence of instructions, software clone detection.**

I. Introduction

STATIC code analysis has a wide range of applications, ranging from categorizing programs [1] and plagiarism detection in homework assignments [2], to the detection of potential defects [3]. While mostly done on the level of source code, some approaches analyze binary or assembler code. This can be done for various reasons, one of which being the comparison of two executables and estimating their similarity. Comparing executables can be done in order to detect malware [4], reverse-engineer software patches [5], detecting license infringement [6], etc. Whatever the goal of the different approaches, they all have to deal with the problem that different compilers can transform the same piece of source code into substantially different binary representations, effectively producing software clones.

Paper received May 13, 2020; accepted June 21, 2020. Date of publication July 31, 2020. The associate editor coordinating the review of this manuscript and approving it for publication was Prof. Miroslav Lutovac.

This work was co-funded by the Ministry of Education, Science, and Technological Development of the Republic of Serbia [III44009 and TR32047].

Nikola Pejić is with the School of Electrical Engineering, University of Belgrade, Bul. kralja Aleksandra 73, 11120 Belgrade, Serbia (e-mail: nikolapeja6@gmail.com).

Miloš Cvetanović is with the School of Electrical Engineering, University of Belgrade, Bul. kralja Aleksandra 73, 11120 Belgrade, Serbia (phone: 381-11-3218385; e-mail: cmilos@etf.bg.ac.rs).

Zaharije Radivojević is with the School of Electrical Engineering, University of Belgrade, Bul. kralja Aleksandra 73, 11120 Belgrade, Serbia (phone: 381-11-3218392; e-mail: zaki@etf.bg.ac.rs).

In general, there are different approaches for comparing binaries on different levels (whole binaries, procedures, and basic blocks), but at the lowest level this is carried out by comparing sequences of instructions, either directly (thus preserving the ordering of the instructions), or by extracting different statistical metrics about the instructions. The benefit of comparing executables from a statistical perspective lies in their resilience to instruction reordering. However, by not considering the ordering of the instructions, the semantic meaning of the sequence is lost, as two completely different sequences of instructions can have the same statistical footprint.

This paper proposes an approach for estimating the similarity of two procedures by calculating the relative Levenshtein distance [7] of their sequences of instructions. The second section contains an overview of the existing methods for comparing assembler procedures. The third section describes the proposed approach, while its evaluation against the existing approaches is given in the fourth section. The fifth section is the conclusion where a brief overview of the results is given, with a few notes on possible improvements.

II. Existing approaches

In this section, a brief summary of five existing approaches is presented. One of them is exclusively a statistical approach, two analyze sequences of instructions, while the rest represent a mixture of approaches (extracting statistical metrics on subsequences).

A. STAT

Stojanović et al. proposed an approach for estimating similarity of procedures based on a variety of metrics [6]. The metrics vary from scalar ones (e.g. frequency of call instructions) to vector ones (e.g. number of occurrences of each instruction), and after they are extracted a comparator function (one for scalar and two for vector metrics) is used to produce a vector of similarity metrics between two procedures. After *transforming* (normalizing) the similarity vectors, a *formula* is used to produce a single value representing the similarity between two procedures.

B. STAT+

Radivojević et al. proposed a set of techniques [8] for improving the performance of the previous method, which consist of:

- omitting stack instructions – as different compilers have different ways of interacting with the stack, those instructions usually add noise and do not hold much information, and as such they can be omitted.
- omitting data transfer instructions – as different compilers use different approaches for register

allocation, the instructions that only move data between registers (or between register and memory) can increase the difference between the output of different compilers for the same source. That is why they could be viewed as noise and omitted from consideration.
- statistics on sequences of instructions – instead of extracting statistics of individual instructions, statistics of subsequences of instructions with fixed length are used when evaluating the similarity.
- procedure inlining – rather than making a call, some compilers decide to inline the referenced procedure, which can drastically affect the similarity estimation of procedures that represent semantic clones. That is why for every procedure call, the similarity metrics from the called procedure are added to the metrics of the caller, thus simulating the inlining.
- similarity threshold – even though two procedures are different, they can have similar metrics, as the metrics used by this approach are statistical. In order to remove false positives, a threshold in the similarity of instruction sets is defined, which if not met immediately marks the two procedures as different.

C. SEQ1

Davis et al. proposed an approach for detecting software clones in assembler code [9] by finding the longest subsequences of instructions that are shared between two procedures. In general, instructions are matched if they are of the same type and if their arguments match. First, for two procedures A and B an instruction that is present in both is located – let that be A_i and B_j. Starting from that pair, the score of the matching increases or decreases depending on whether or not the succeeding instructions match. When a mismatch happens, the next instruction pairs that are considered are A_{i+p} and B_{j+q}, where $p, q \in N$. If the score of the matching becomes negative, the matching resets and starts off with the next pair of matching instructions, but not before an optimization of the previous sequence is carried out in an attempt to increase its score. The highest matching score for a pair of procedures is taken as their degree of similarity.

D. SEQ2

Pejić et al. proposed an approach for estimating the similarity between procedures via neural networks [10]. The approach extracts opcodes from the procedures and encodes them into integers, making sure to map opcodes that perform the same action (e.g. all branch instructions; all instructions that perform addition - `ADD`, `ADDB`, `ADDW`, etc.) into the same value. Next, the sequence of integers is used as input for an LSTM network, which outputs a sequence of features of the same length. A DNN takes the output from the previous phase and generates a predefined number of metrics. The vectors of metrics from the two procedures are then compared and normalized into a single value representing the similarity between the two procedures.

E. SEQ-STAT

Santos et al. proposed an approach [4] for detecting malware by analyzing sequences of opcodes and assessing the similarity against known malware. Rather than considering whole instructions, a sequence of opcodes is extracted from a given procedure. The frequency of all possible subsequences of fixed length is calculated using weighed term frequency, which uses a predefined set of weights for each opcode. Finally, similarity between two procedures is determined by calculating the cosine similarity between the weighed term frequency vectors.

III. Proposed Approach

In order to reduce the noise compilers can generate (caused by different register allocation policies, different optimization techniques, etc.), it was decided to consider procedures only as sequences of opcodes. Now, the problem of comparing two procedures is reduced to the comparison of sequences of tokens, which is why it was decided to use a technique from the field of information theory. The Levenshtein distance [7] represents a similarity metric between two sequences of characters and is defined as the minimum number of insertions, deletions or substitutions needed to transform one of the sequences into the other.

A schematic overview of the approach proposed in this paper is given in Fig. 1 and consists of the following phases:

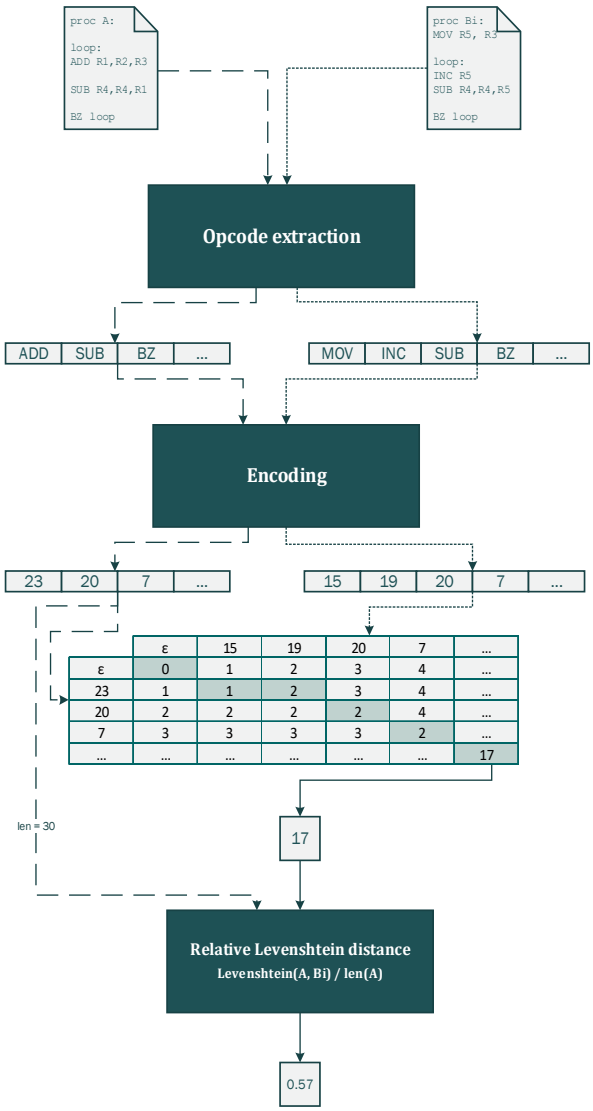
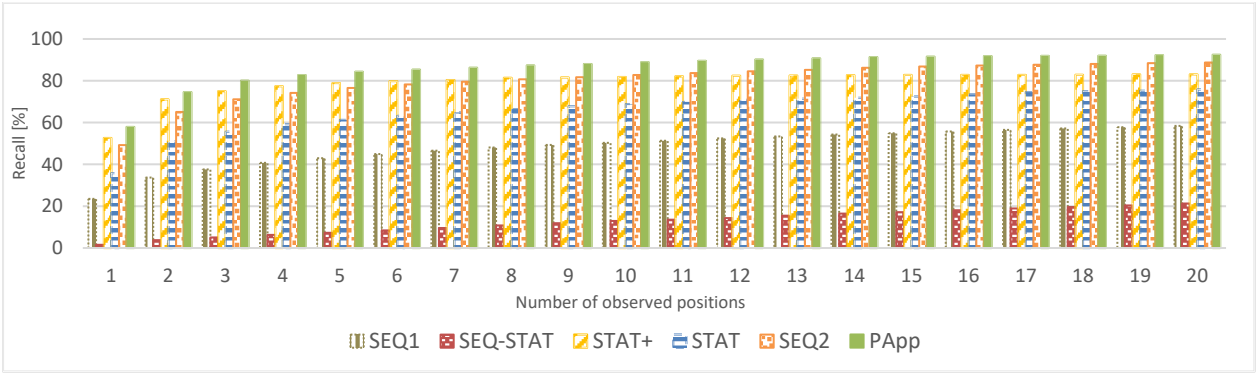
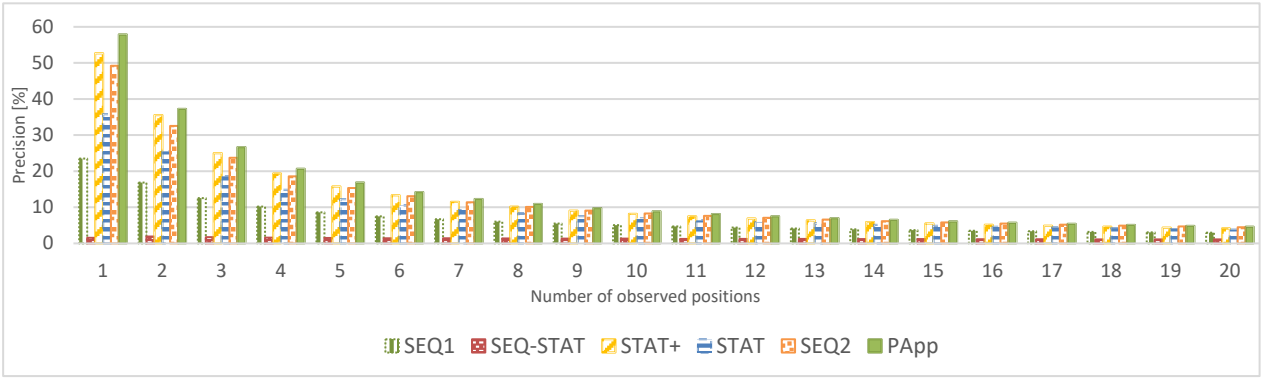


Fig. 1. Schematic representation of the proposed approach.



(a)



(b)

Fig. 2. Comparison between the proposed approach and existing methods in terms of recall (a) and precision (b) for the first 20 observed positions.

- opcode extraction – extracts opcode tokens from a procedure given in assembler form.
- encoding – maps the opcode tokes into integer values. For this phase, the same semantic mapping was used as in [10], mapping instructions that perform highly similar actions into the same value.
- Levenshtein distance – the Needleman–Wunsch algorithm [11] is used to calculate the number of operations needed to transform the encoded sequence of one procedure into the other.
- relative Levenshtein distance – although completely valid, the Levenshtein distance can return values from the set of natural numbers. In order to have better interpretability of the similarity metric, the relative Levenshtein distance is calculated by dividing the output of the previous phase with the length of the first sequence (procedure A).

IV. Evaluation

The approach described in the previous section was implemented in the Python 3 [12] programing language. The evaluation itself is carried out in the same way as in [10]:

- The dataset consists of five programs (bayes, genome, intruder, ssca2, and vacation) from the STAMP benchmark [13], that were compiled with five different compilers (Keil v4.60.0 [14], IAR v6.50.2 [15], CodeSourcery v4.7.2 [16], CrossWorks v2.3.0 [17], and SysProgs v4.6.3 [18]) with the `-O3` flag.
- A single test case consisted of a randomly selected ordered pair $(A, \mathcal{B})$, where A is a single procedure from the dataset, and $\mathcal{B}$ is a set of procedures from one of the programs. All procedures from $\mathcal{B}$ were compiled using the same randomly selected compiler (the selection of that compiler was not influenced by the compiler used to generate A). There was exactly one $\bar{\bar{B}} \in \mathcal{B}$ that originated from the same source code as A.
- The evaluation of one test case consisted of evaluating the similarity of A with every procedure from $\mathcal{B}$ and then ranking the procedures from $\mathcal{B}$ in a decreasing order of their similarity to A.
- The performance was measured by measuring precision and recall in terms of the number of observed procedures.
- The whole evaluation consisted of 3300 test cases.

Fig. 2 shows the achievements of the proposed approach compared to existing ones. As the dataset and experimental parameters were the same as in [10], the evaluation of the existing methods was not repeated. As it can be seen, the proposed approach achieves higher recall and precision than any of the existing approaches that were considered, having between 3.59% and 9.45% (on average 6.06%) higher recall, and between 0.17% and 4.73% (on average 1.33%) higher precision than the SEQ2, when considering the first 20 ranked procedures. When compared against the STAT+, it achieved between 3.45% and 9.53% (on average 7.23%) higher recall, and between 0.45% and 1.72% (on average 1.00%) higher precision. The difference in the achievements can better be seen in the precision-recall diagram shown in Fig. 3.

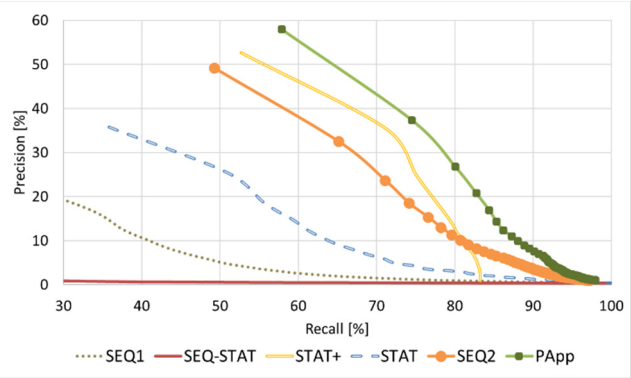


Fig. 3. Precision-recall curves for the evaluated methods.

V. Conclusion

This paper proposes an approach for comparing assembler procedures as sequences of opcodes based on the Levenshtein distance. Evaluation showed that the proposed approach outperforms the existing approaches against which it was evaluated, having on average around 6% higher recall than the second best. However, as the evaluation was carried out over a single dataset, further tests using different datasets and against other methods need to be carried out, in order to more precisely measure the performance of the proposed approach and its significance.

One of the possible improvements could be the adoption of some of the techniques used in STAT+, like omitting instructions that hold little information about the procedure. Another possible improvement lies in the fact that the basic version of the Levenshtein distance considers all insertions, deletions, and substitutions to be of equal weight. If, for example, a multiplication instruction is changed into a logical shift left, then the difference isn't as significant as if the multiplication is changed into a logical and. In order to adapt to these situations, a custom scoring matrix should be defined where common compiler transformations are given lower weights in order to decrease their impact on the final score. With some of these improvements, the performance of the proposed approach could be further improved, making it an even more relevant approach.

References

[1] L. Li *et al.*, "Static analysis of android apps: A systematic literature review," *Information and Software Technology*. 2017, doi: 10.1016/j.infsof.2017.04.001.

[2] M. Misic, Z. Sustran, and J. Protic, "A comparison of software tools for plagiarism detection in programming assignments," *International Journal of Engineering Education*, vol. 32, no. 2, pp. 738–748, 2016.

[3] F. Zampetti, S. Scalabrino, R. Oliveto, G. Canfora, and M. di Penta, "How Open Source Projects Use Static Code Analysis Tools in Continuous Integration Pipelines," in *IEEE International Working Conference on Mining Software Repositories*, 2017, doi: 10.1109/MSR.2017.2.

[4] I. Santos *et al.*, "Idea: Opcode-sequence-based malware detection," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2010, doi: 10.1007/978-3-642-11747-3_3.

[5] T. Dullien and R. Rolles, "Graph-based comparison of executable objects (english version)," *Sstic*, 2005, doi: 10.1.1.96.5076.

[6] S. Stojanović, Z. Radivojević, and M. Cvetanović, "Approach for estimating similarity between procedures in differently compiled binaries," *Information and Software Technology*, 2015, doi: 10.1016/j.infsof.2014.06.012.

[7] V. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Physics Doklady*, 1966.

[8] Z. Radivojević, M. Cvetanović, and S. Stojanović, "Comparison of binary procedures: A set of techniques for evading compiler transformations," *Computer Journal*, 2015, doi: 10.1093/comjnl/bxv076.

[9] I. J. Davis and M. W. Godfrey, "From whence it came: Detecting source code clones by analyzing assembler," in *Proceedings - Working Conference on Reverse Engineering, WCRE*, 2010, doi: 10.1109/WCRE.2010.35.

[10] N. Pejic, M. Cvetanovic, and Z. Radivojevic, "Estimating similarity between differently compiled procedures using neural networks," in *27th Telecommunications Forum, TELFOR 2019*, 2019, doi: 10.1109/TELFOR48224.2019.8971103.

[11] S. B. Needleman and C. D. Wunsch, "A general method applicable to the search for similarities in the amino acid sequence of two proteins," *Journal of Molecular Biology*, 1970, doi: 10.1016/0022-2836(70)90057-4.

[12] Python Software Foundation, "Welcome to Python.org," 2001. https://www.python.org/.

[13] C. M. Chí, J. W. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *2008 IEEE International Symposium on Workload Characterization, IISWC'08*, 2008, doi: 10.1109/IISWC.2008.4636089.

[14] Arm Limited, "Embedded Development Tools." http://www.keil.com/.

[15] IAR Systems, "IAR Embedded Workbench." https://www.iar.com/iar-embedded-workbench.

[16] Mentor, "Sourcery CodeBench," [Online]. Available: https://www.mentor.com/embedded-software/sourcery-tools/sourcery-codebench/overview.

[17] Rowley Associates Ltd, "CrossWorks for ARM." https://www.rowley.co.uk/arm/index.htm.

[18] Sysprogs, "Prebuilt Windows Toolchain for ARM." http://gnutoolchains.com/arm-eabi/.