Information and Software Technology 58 (2015) 259–271

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Approach for estimating similarity between procedures in differently compiled binaries

Saša Stojanović, Zaharije Radivojević, Miloš Cvetanović [*]

School of Electrical Engineering, University of Belgrade, Bulevar kralja Aleksandra 73, 11000 Belgrade, Serbia

ARTICLE INFO

Article history:
Received 6 July 2013
Received in revised form 9 May 2014
Accepted 22 June 2014
Available online 2 July 2014

Keywords:
Software clone
Clone detection
Semantic clone
Software metric
Binary code analysis

ABSTRACT

Context: Detection of an unauthorized use of a software library is a clone detection problem that in case of commercial products has additional complexity due to the fact that only binary code is available.
Objective: The goal of this paper is to propose an approach for estimating the level of similarity between the procedures originating from different binary codes. The assumption is that the clones in the binary codes come from the use of a common software library that may be compiled with different toolsets.
Method: The approach uses a set of software metrics adapted from the high level languages and it also extends the set with new metrics that take into account syntactical changes that are introduced by the usage of different toolsets and optimizations. Moreover, the approach compares metric values and introduces transformers and formulas that can use training data for production of measure of similarities between the two procedures in binary codes. The approach has been evaluated on programs from STAMP benchmark and BusyBox tool, compiled with different toolsets in different modes.
Results: The experiments with programs from STAMP benchmark show that detecting the same procedures recall can be up to 1.44 times higher using new metrics. Knowledge about the used compiling toolset can bring up to 2.28 times improvement in recall. The experiment with BusyBox tool shows 43% recall for 43% precision.
Conclusion: The most useful newly proposed metrics are those that consider the frequency of arithmetic instructions, the number and frequency of occurrences for instructions, and the number of occurrences for target addresses in calls. The best way to combine the results of comparing metrics is to use a geometric mean or when previous knowledge is available, to use an arithmetic mean with appropriate transformer.

1. Introduction

The pressure the market lays on the software industry implies a need to reuse a software code and libraries [1]. In order to gain a greater share of the market, software companies and engineers make their code available to others [2]. The code can be made available, free of charge, for a period of time (e.g. during development), but after this period expires, a suitable fee is expected (e.g. during production). A real-world motivating scenario for this paper is the reuse of a software library in a commercial product without an appropriate permission from the owner of the library. A process of collecting evidences about the reuse is an activity of finding a searched procedure that originates from the software library, among target procedures extracted from the commercial product. The target procedures are most likely available only as a binary code. The binary code is a result of compilation with an unknown toolset. Consequently, the searched procedure can appear in a set of target procedures in a different, but semantically equivalent form, known as a software clone.

Each of the differences between binary codes, introduced by toolsets, can be considered as one out of four software clone types. Two different toolsets can possibly translate a fragment of a source code into exactly the same binary codes, namely software clones type 1. Even though the same binary codes are produced by a compiler, if mapped differently by a linker, the final binary codes can represent syntactically identical copies with potentially renamed literals, or software clones type 2. Moreover, a compiler can introduce modifications such as an instruction reorder or a recognition of a dead code, leading to software clones type 3, that are defined as copies with minor modifications such as changed, added, or removed instructions. Similarly, the compiler can detect code fragments that can be optimized if replaced by syntactically different

[*] Corresponding author. Tel.: +381 113218385; fax: +381 113248681.
E-mail addresses: stojsasa@etf.bg.ac.rs (S. Stojanović), zaki@etf.bg.ac.rs (Z. Radivojević), cmilos@etf.bg.ac.rs (M. Cvetanović).

http://dx.doi.org/10.1016/j.infsof.2014.06.012
0950-5849/© 2014 Elsevier B.V. All rights reserved.

code fragments that perform the same computation, named as software clones type 4.

A clone detection is a technique that helps identify procedures that are potential software clones [3]. The detection process comprises of two steps: the first one is to determine similarity between considered procedures and the second one is to decide whether considered procedures are actually clones. In the context of the given scenario, the clone detection needs to be flexible and to weight a recall more than a precision because a user is always willing to tolerate some falsely identified clones in order to find the true one. One way of achieving the flexibility of a clone detection is to rank all the target procedures as potential clones based on similarities with the searched procedure and let the user to make a final decision.

This paper presents an approach, based on software metrics, for estimating similarities between a procedure from one binary code and a set of procedures from another binary code. Moreover, the set of procedures is ranked according to estimated similarities in order to find possible clones. The approach proposes new software metrics obtained at the level of procedures, which take into account syntactical changes introduced by different compiler optimizations (clone types 2, 3 and 4). Moreover, the paper evaluates different ways of combining new and existing metrics, and examines the contribution to the similarity detection success of each combination. The evaluation is conducted through four experiments. The first three experiments involve about 300,000 compared procedure pairs, that originate from 5 benchmark programs with over 1300 procedures in total, compiled with 5 different compilers using O3 optimization level without debug information, and O0 optimization level without debug information. The fourth experiment, based on a real world program, uses the same setup extended with optimizations for size and speed and involves over 1,500,000 compared procedure pairs.

The evaluation shows that adding proposed metrics to the existing ones increases recall. Moreover, the experiments show that recall depends on the used compiler and its options. The evaluation also shows that in comparison with some existing clone detection tools, the proposed approach achieves higher recall without degrading the precision.

The rest of the paper is organized as follows. Section 2 gives the problem statement, assumptions, and research questions that will be examined in the following sections. Section 3 provides a brief overview of the existing techniques and representative tools potentially applicable for detection of the software reuse in binary codes. The proposed approach is described in Section 4, while the conducted experiments and the empirical evaluation are presented in Section 5. Section 6 concludes the paper.

2. Problem statement and assumptions

The scenario described in the introduction section can be stated as a problem of estimating similarities between procedure A and each procedure B_i from set B, and ranking procedures from set B according to the estimated similarities. This paper considers the problem under following assumptions: AS1. Procedures included in set B are available only in binary code; AS2. The binary code may not contain the symbol tables; AS3. Exactly one procedure B_i has the same source code as procedure A; AS4. The differences between procedure A and the corresponding procedure B_i come from the use of different toolsets for compiling and linking processes.

This paper proposes an approach that uses existing software metrics, introduces new metrics, and defines how to combine them. The approach is evaluated in terms of precision, recall and a single measure that trades off between them, F-measure. The paper explores three different research questions: RQ1. Does adding new metrics to the existing metrics contribute to the recall in first N ranked procedures? RQ2. Does the recall in the first N ranked procedures achieved by the proposed approach depend on a compiler, optimization level, and a problem context? RQ3. Does the proposed approach achieve better results than the existing tools in terms of precision, recall, and F-measure?

3. Related work

Comparing procedures can be viewed from two different aspects, when procedures originate from mostly identical or mostly different binary codes. For example, in order to detect potential vulnerabilities that might be exploited by malwares, the original binary and the patched version of the same binary are compared. However, in order to find similar procedures when mostly different binary codes are compared, clone detection can be applied on a level of procedures.

During vulnerability analysis, the differences between binary codes introduced by compilers are of special interest. For example, the framework described in [4] considers a register allocation, instruction reordering and branch inversion. The framework creates a bijective mapping between the procedures for two compared executables by improving iteratively a partial graph isomorphism on the call-graphs of the executables, and after that it does the same for basic blocks and flow-graphs. Another example is malware analysis tool, BinSlayer [5], which besides a register allocation and instruction reordering considers junk/do-nothing code and obfuscating. BinSlayer calculates the differences between two binaries by using algorithm that fuses the BinDiff algorithm with the Hungarian algorithm for a bi-partite graph matching. On the other hand, determining the similarities of executables in the presence of obfuscating, addressed in [6], is done by computing frequencies of opcodes and opcode sequences and by calculating a discrimination ratio.

Considering clone detection shows that a large number of tools are developed [7,8]. The tools can be evaluated according to different properties, but from the aspect of this paper, the applicability on a binary code and tolerance on the modifications introduced by compilers are the most important ones. The applicability on a binary code depends on a tool's support for the low level languages or a possibility of adaptation for such a support reflected through language dependency and a needed effort. Tolerance on modifications introduced by a compiler mostly depends on a detection technique used by a tool. Table 1 summarizes the properties of the tools that depend on used detection technique and lists some representative tools according to the results presented in the surveys [9–11].

Most of the tools described in the literature listed below are designed for the high level languages. But, their applicability on binary code, or on an equivalent assembly code, requires some adaptations at least on the input side for all tools, except for most of the text based ones. A new lexer is needed for token based tools and for most of the metrics based ones. Beside the new lexer, the tools based on abstract syntax tree (AST) and program dependency graph (PDG) also require a new parser. Depending on the used metrics, a simplified parser may also be required in the case of metrics based tools. Instead of aforementioned adaptations, the appropriate decompiler could be used. Due to complexities involved in decompilation process, the existing decompilers often produce source code that cannot be compiled again, which may be an issue for some clone detection tools.

The surveys also investigate the tools' tolerance on different types of modifications. Most of the existing tools are able to detect software clones of type 1. Detection of type 2 software clones is