A Method to Evaluate Binary Code Comparison Tools

Mariam Arutunian
System programming department
Russian-Armenian University
Yerevan, Armenia
arutunian@ispras.ru

Hripsime Hovhannisyan
System programming department
Russian-Armenian University
Yerevan, Armenia
hr.hovhanisyan@ispras.ru

Vahagn Vardanyan
System programming department
Russian-Armenian University
Yerevan, Armenia
vaag@ispras.ru

Sevak Sargsyan
System programming department
Russian-Armenian University
Yerevan, Armenia
sevaksargsyan@ispras.ru

Shamil Kurmangaleev
System programming department
Institute for System Programming of the Russian Academy of Sciences
Moscow, Russia
kursh@ispras.ru

Hayk Aslanyan
System programming department
Russian-Armenian University
Yerevan, Armenia
hayk@ispras.ru

***Abstract*— Binary code comparison tools are widely used to analyze vulnerabilities, search for malicious code, detect copyright violations, etc. The article discusses the best three tools known at the time - BCC, BinDiff, Diaphora. They are based on static analysis of programs. The tools receive as input data two versions of the program in binary form and match their functions. The purpose of the article is to assess the quality of the tools. We developed a testing system to automatically determine the precision and recall of each instrument. F_1 score on the developed testing system for BCC instrument is 85.6%, for BinDiff - 82.4%, for Diaphora - 64.7%.**

Keywords*— *binary code analysis, BCC, BinDiff, Diaphora

I. Introduction

Binary code comparison tools compare two versions of programs to determine their similarities and differences. The comparison can be done at the level of basic blocks, functions, or entire programs. Often the source code of a program is not available, thus binary code analysis has fundamental importance.

Revealing the similarities and differences of executable files is a difficult task since the compilation process can remove (depending on optimizations) some information of the program including variable names, function names, and data structure definitions. Moreover, the resulting binary can be changed significantly when source code is compiled with different compilers and different optimizations, as well as, for the different target operating systems and architectures.

Programs' binary code comparison has a wide range of applications, such as bug detection, malicious program identification, automatic patch generation, patch analysis, software copyright infringement detection, etc.

There are many works devoted to the analysis of program changes. This article provides an overview and comparison of BinDiff [1] [2], Diaphora [3], and BCC [4] tools, which are currently supported and are showing the best results. In order to evaluate and compare the tools, a method for a testing system is developed and implemented.

II. Binary Code Comparison Tools Overview

BinDiff tool (version 6) [1] [2] implements different metrics for function mapping. Metrics are calculated on control flow graphs and function call graphs. In their paper, the authors propose a method to calculate MD-index [5] hashes for graphs. Further MD-index is used to calculate metrics on those graphs. Two functions are matched if their metrics are equal and unique.

Diaphora tool (version 2.0.3) [3] matches functions using various sets of heuristics on control flow graphs. Heuristics are applied sequentially; if there are still unmached functions after applying some heuristics, another one is applied.

BCC tool [4] is divided into two stages. At the first stage, function call graphs and program dependence graphs (PDG) are generated. At the second stage, functions are compared using the generated graphs. The algorithm, which matches functions consists of two main steps. Functions are first matched using a series of heuristics. Those functions, which were not matched using heuristics are matched using the algorithm for determining the maximum common subgraph of PDG.

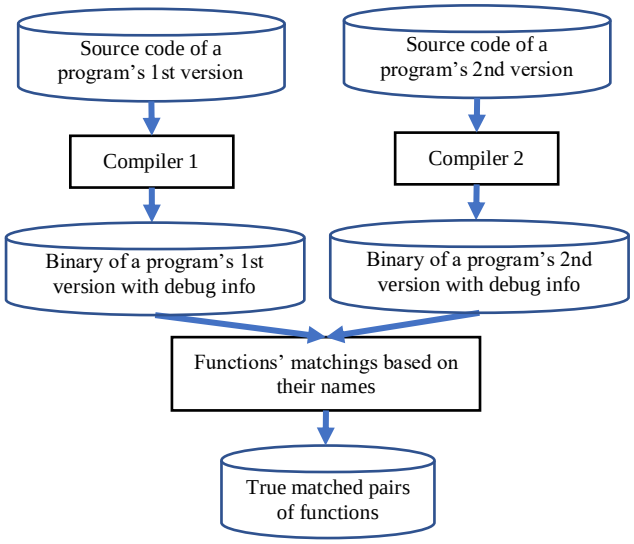

Fig. 1. Automated test generation

III. Test System Scheme

In order to evaluate binary code comparison tools, a method for a testing system is developed. Tools are evaluated by precision, recall, and F_1 score. They are defined as follows:

$$precision = \frac{tp}{tp + fp}$$

$$recall = \frac{tp}{tp + fn}$$

$$F_1\ score = 2 * \frac{precision * recall}{precision + recall}$$

where tp is true positive results count, i.e., function's pairs count, which the tool detected and they are also in true matched pairs set of functions. Oppositely, fp is false positive results count, i.e., function's pairs count, which the tool detected and they are not in true matched pairs set of functions. fn is function's pairs count that tool has not detected, but should.

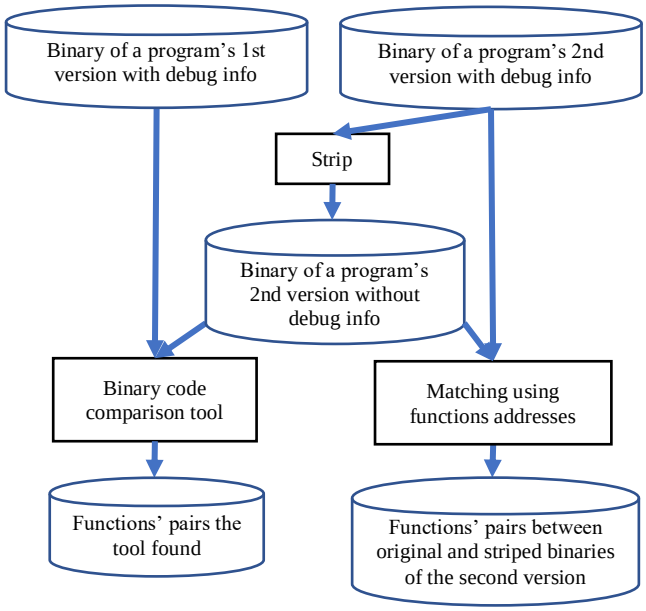

Fig.2. Running binary code comparison tools on generated tests

The testing system consists of three parts - automatic test generation, running binary code comparison tools on the generated tests, and their evaluation using generated tests.

Fig. 1 shows the scheme of automatic test generation. Two versions (not necessarily different) of the same program are used to generate tests. At first, the source code of both versions is compiled with debug information. Compilation can be done with different optimization flags. Debug information allows using the names of the functions. If two functions from the first and the second versions of the program have the same name, they are matched (true matched pairs of functions). Ideally, binary code comparison tools should find the same set of functions' matches on the same inputs.

In the second part of the testing system (Fig. 2), the binary of the second version is striped: symbols are removed from it (original function names are also removed) using the strip utility from coreutils [6]. Then the binary of the first version and the stripped binary are passed to a binary code comparison tool to get functions' matches.

The purpose of removing symbols from a binary file is to make the generated tests as close as possible to real examples, since such information is almost always not available in analyzing programs.

Additionally, we get function matches between striped and original binaries of the program's second version. As after removing symbols from functions addresses stay the same, we match those functions using their addresses.

The third part of the testing system (Fig. 3) evaluates binary code comparison tools. It uses information from previous parts to calculate precision, recall, and F_1 score.

The testing system automatically parses the results of the instruments and brings them to a single form. For each found pair of functions (f, f'), the original name of f' is restored using information about functions' pairs between original and striped binaries of the second version. Let's suppose the original name

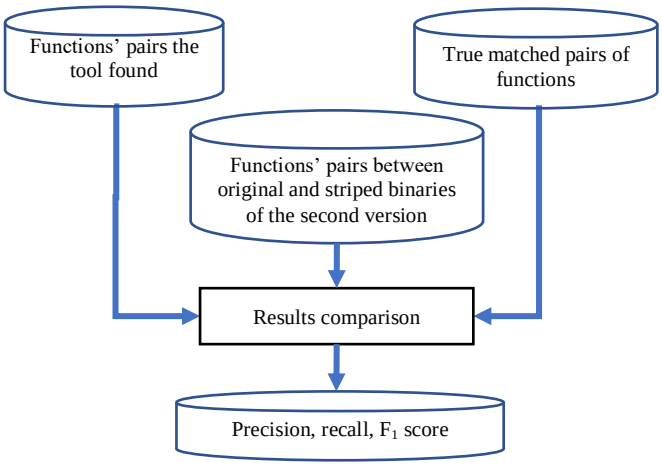

Fig.3. Calculation of tool's precision, recall, and F_1 score

of f' is g. If (f, g) is in the set of true matched pairs of functions, then the result is considered true positive; otherwise, it is a false positive. If there are results in true matched pairs of functions, that the tool didn't detect, they are considered as false negatives. Precision, recall, and F_1 score are calculated based on true positives, false positives, false negatives count.

IV. Results

We obtained results using 105 programs included in coreutils set [6]. Two versions of the same program source code were passed to the testing system. Also, source files were compiled by g++ (version 9.2.1) [7] and clang++ (version 9.0.0) [8] compilers with different optimization flags for the x86-64 architecture. Results of BCC, BinDiff (version 6), and Diaphora (version 2.0.3) tools are in Table 1.

The tools' precision and recall are lower if programs are compiled with different optimization flags. It shows that compilers generate extreamly different binary code in the case of different optimizations. Even the results with different compilers, but the same optimization are higher.

We can see from the table that BCC tool's results exceed competitors. The best accuracy shows Diaphora, but the recall of the tool is much smaller than the others' recall. On average, F_1 score for BCC is 85.6%, for BinDiff - 82.4%, for Diaphora - 64.7%. Moreover, the difference between F_1 scores is greater, when the difference between the versions of the analyzed programs is larger (there are many changes), or when they are compiled with different compilation flags.

TABLE I. Tools results

Version 1, compiler, optimizations	Version 2, compiler, optimization, stripped	BCC precision (%)	BCC recall (%)	BinDiff precision (%)	BinDiff recall (%)	Diaphora precision (%)	Diaphora recall (%)
8.30 clang++ o0	8.30 clang++ o0	98.8	98.3	98.8	97.9	98.3	75
8.30 g++ o0	8.30 g++ o0	98.8	98.3	98.8	97.9	98.3	74.7
8.30 clang++ o2	8.30 clang++ o2	97.2	85.6	98.3	85.9	97.5	59.2
8.30 g++ o2	8.30 g++ o2	98.2	82.4	98.2	81.7	97.3	55.2
8.30 clang++ o0	8.30 clang++ o2	79.3	67.4	72.2	61.5	93.1	34.3
8.30 g++ o0	8.30 g++ o2	84.8	69.2	75.5	63.7	91.3	34.5
8.30 clang++ o2	8.30 clang++ o3	90.7	81.2	86.6	76.6	97.2	41.2
8.30 g++ o2	8.30 g++ o3	90	77	88.7	75.4	93.5	44.8
8.30 g++ o0	8.30 clang++ o0	94.9	89.4	91.2	85.4	95.6	38.1
8.30 g++ o2	8.30 clang++ o2	89.1	76.1	81.1	71	93.8	36.7
8.30 g++ o0	8.30 clang++ o2	78.9	67.7	71.2	61.5	93	34.5
8.29 g++ o0	8.30 g++ o0	97.8	98.2	97.8	97.8	97.6	72.5
8.29 g++ o2	8.30 g++ o2	97	82.2	96.6	81.4	96.3	52.8
7.6 g++ o0	8.30 g++ o0	72.1	69.5	62.9	62.9	92.5	35.5
7.6 g++ o2	8.30 g++ o2	85.4	77.9	83.4	76.5	95.1	46.5
Average		**90.2**	**81.4**	**86.8**	**78.5**	**95.4**	**49.0**

V. Conclusion

The article presents a method to evaluate binary code comparison tools. These tools are chosen because they are supported until now, are widely used, and show the best results. The comparison was carried out on several coreutils' programs. The results show that the precision, recall, and F_1 score of the tools are high when the difference between two versions of the analyzed program is smaller, and when they are compiled with the same compiler optimizations. The results of the tools are competitive, but on average, BCC shows the best F_1 score.

Acknowledgment

This work was supported by the RA Science Committee and Russian Foundation for Basic Research in the frames of the joint research project SCS 20RF-033 and RFBR 20-57-05002 accordingly.

References

[1] T. Dullien and R. Rolles, "Graph-based comparison of executable objects," *Symposium sur la Securite des Technologies de l'Information et des Communications*, 2005.

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

[3] J. Koret. https://github.com/joxeankoret/diaphora.

[4] H. Aslanyan, A. Avetisyan, M. Arutunian, G. Keropyan, S. Kurmangaleev, and V. Vardanyan, "Scalable Framework for Accurate Binary Code Comparison," in *2017 Ivannikov ISPRAS Open Conference (ISPRAS)*, Moscow, 2017.

[5] T. Dullien, E. Carrera, S. Eppler, and S. Porst, "Automated attacker correlation for malicious code," DTIC Document, 2010.

[6] "Coreutils - GNU core utilities," https://www.gnu.org/software/coreutils/.

[7] "GCC, the GNU Compiler Collection," https://gcc.gnu.org/.

[8] "Clang: a C language family frontend for LLVM," https://clang.llvm.org/.