University of Sindh Journal of Information and Communication Technology (USJICT)
Volume 6, Issue 1, March 2022
ISSN-E: 2523-1235, ISSN-P: 2521-5582
Website: http://sujo.usindh.edu.pk/index.php/USJICT/

Exploring and Assessing the Trivial Compiler Equivalence

Habibullah Nangraj, Gulsher Laghari
Institute of Mathematics and computer science, University of Sindh, Jamshoro.
nangrajhabib@gmail.com, gulsher.laghari@usindh.edu.pk

Abstract: Mutation testing is the state-of-the-art technique to assess the fault-detection capability of a test suite. However, its adoption in industry is deterred by few of its inherent limitations including the equivalent mutants. Since the equivalent mutants are functionally similar to the original program, the test suite cannot kill them, hence they produce false alarms for the developers and reduce the mutation score. Although to automatically verify whether the mutant is equivalent to the original program is undecidable, yet there exist heuristics such as trivial compiler equivalence to automatically eliminate sufficient equivalent mutants. In this paper, we explore the use of compiler optimizations at assembly level code to detect equivalent mutants and find that it can indeed detect equivalent mutants.

Keywords: —Mutation testing, equivalent mutants, trivial compiler equivalence

I. INTRODUCTION

Software testing is a quintessential activity in modern software development to ensure software quality by eliminating potential faults from sneaking into production.

Thus, a significant proportion of the code comprises the test code— the code that automates the whole testing and serves as the safety-net for the production code. Since the test code is used to test the production code, the natural question to ask is what tests the test code? More precisely, how to test fault detection capability of the test code?

The answers to these questions are provided by mutation testing. Indeed, mutation testing is used to help assess the test suite effectiveness in detecting the faults [1]. Mutation testing simply injects artificial faults (in the form of minor syntactic changes) into production code and puts the test suite to the test to detect those faults.

The program with minor syntactic change is called the mutant. If the test suite fails on the mutant, it essentially kills the mutant implying that it is reliable and can detect the faults. Contrarily, if the test suite passes, the mutant is said to have survived implying that the test suite needs improvement. The test suite effectiveness in terms of its ability to detect faults is measured via the metric mutation score [2]. It is simply the percentage of the mutants killed.

Mutation testing is slowly being adopted in industry [3]–[5]. However, its widespread adoption is largely deterred by its inherent cost limitations, such as the presence of equivalent mutants.

An equivalent mutant is the program variant created by mutation tool that is syntactically different from the original program, yet it is semantically like the original program. Since the equivalent mutants are functionally same to the original program, the test suite cannot kill them. Therefore, the presence of large number of equivalent mutants produces the false alarms unnecessarily reducing the mutation score— they must be discarded from mutation analysis!

While manual verification of equivalent mutants is not feasible [6] and automatically detecting them is undecidable [7], there are heuristics to automatically eliminate them, such as trivial compiler equivalence (TCE) [8]

TCE generates optimized binary files of both the original program and the mutant, compares them together. If these two are the same, the generated mutant is declared as equivalent.

In this paper, we further explore the use of optimized assembly level code rather than binary code to detect equivalent mutants.

We organize the paper as follows. Section II provides the background information and the closely related work. Section III provides experimental details and Section IV provides the results. Finally, Section V concludes the paper.

II. BACKGROUND AND RELATED WORK

Trivial compiler equivalence (TCE).

TCE (as the name suggests) is a trivial technique proposed by Papadakis et al. that exploits compiler optimizations to detect equivalent mutants [8]. In TCE, the original program and the mutant generated by a mutation tool (such as *MuCPP*) are simply compiled into binaries by applying a specific optimization technique provided by the compiler (such as *gcc*). Then, the two binary files are compared to check if they are the same or not? If the two files are the same, the generated mutant is equivalent mutant.

Baldwin and Sayward were the first to suggest that since the compiler optimization tries to produce a syntactically different yet semantically the same version of the original program, such optimization can also be used to detect an equivalent mutant [9].

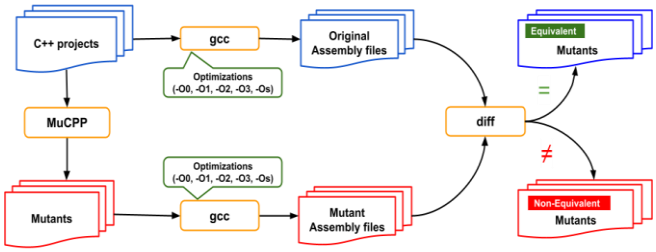

Fig. 1: Overall experimental setup of detecti

However, Papadakis et al. recently formalized the use of compiler equivalence and named the approach as TCE [8]. Since then, TCE has been studied by many researchers.

Delgado-Perez et al. applied TCE to an industrial application to assess its effectiveness [10]. Similarly, others have also found that TCE is reasonably efficient to detect equivalent mutants compared to manual review [8], [11].

Houshmand and Paydar extended the TCE to apply it on Java programs and named the approach as TCE+ [12].

In this paper, we also use TCE but we explore its use at assembly level code rather than binary level code.

III. EXPERIMENTAL SETUP

This section provides the details of the experiment.

Mutation tool. We use MuCPP∗ tool for mutant generation [13]. MuCPP generates the mutants and stores them in git repository. Each mutant is stored in a separate branch.

Compiler optimizations. We use gcc compiler with 5 different optimization options† provided by gcc compiler. These optimizations can be triggered by providing corresponding switch (flag) to gcc while compiling.

- -O0 (the default). Reduces compilation time.
- -O1. Reduces code size and execution time.
- -O2. Optimizes more than -O1.
- -O3. Optimizes even more than -O2.
- -Os. Optimizes for code size.

Each of the switches (flags) above, enable different optimization techniques.

Compiler output. The gcc compiler generally produces compiled code in binary (executable form). However, it can also compile the code in assembly language using -S switch (flag). As we detect equivalent mutants at assembly level code, we compile the C++ code into assembly. It is important to mention that the gcc includes meta data alongside assembly code.

Equivalent mutant detection via TCE. To detect whether a particular mutant generated by MuCPP is equivalent to the original program, we use the Linux utility diff as follows.

TABLE I: Subject programs used in this study.

Program	LOC	Functionality
Quick_Sort	44	Quicksort algorithm
Bubble_Sort	39	Bubblesort algorithm
Insert_Sort	41	Insertionsort algorithm
MinMax	45	Min/Max numbers
Cal	46	Arithmetic operations

Where the *original-file* is the file that contains the optimized assembly code for the original C++ file, while *mutant-file* contains the optimized assembly code for the mutant C++ file. The *diff* program simply compares the contents of these two files and returns the differences, if any.

diff original-file mutant-file

In case, there are no differences between the two assembly files, we mark the mutant as *equivalent* otherwise *non-equivalent.*

We run our experiments on Ubuntu 20.04 LTS with gcc compiler version 7.5.

Subjects programs. We have chosen to use 5 small, yet representative C++ programs for our experiments. Four of these programs (*Quick_Sort*, *Bubble_Sort*, *Insert_Sort*, *MinMax*) have also been used by Huan et al. [14]. While the *Cal* program is written in object-oriented C++ by us. Table I provides the details of the subject programs.

Protocol. Our experimental protocol is quite simple. First, for each subject program in Table I, we generate the mutants with MuCPP. Next, we compile each mutant of a subject program with all the optimization options as described in Section III to produce the optimized assembly code in a file. Likewise, we also compile the original subject program with all the optimization options to produce the optimized assembly code in a file. Finally, we compare the optimized assembly code of both original program and the mutant to determine if the mutant is equivalent or non-equivalent. Figure 1 shows the overall experimental setup. This whole experiment is automated as bash script.

TABLE II: Number of detected equivalent mutants.

		#Equivalent Mutants				
Program	**#Mutants**	**-O0**	**-O1**	**-O2**	**-O3**	**-Os**
Quick_Sort	170	4	16	3	5	3
Bubble_Sort	69	4	5	5	5	5
Insert_Sort	79	2	8	8	9	8
MinMax	74	3	7	7	9	7
Cal	44	1	3	3	3	3
Total	**436**	**13**	**39**	**26**	**31**	**26**

IV. RESULTS

Through our experiments, we could confirm that it is also possible to detect equivalent mutants via trivial compiler equivalence at assembly level code. Our automated script detected and calculated equivalent mutants with all 5 optimizations. We also manually verified that the mutants marked by the script as equivalent were indeed equivalent mutants. Table II presents the number of generated mutants (column # Mutants) and the number of mutants marked as equivalent with different optimizations. We can readily observe that -O3 optimization can help detect more equivalent

mutants for four programs. However, this is not the case for the program *Quick_Sort*. There, -O1 optimization detects way more (about 3x) more equivalent mutants.

Furthermore, -O2 and -Os optimizations exhibit the same behavior, they contribute equally to detecting equivalent mutants. This should come as no surprise, since the -Os optimization essentially enables all -O2 optimizations except few.

V. CONCLUSION AND FUTURE WORK

In this paper, we explored the use of compiler optimizations at assembly level code via trivial compiler equivalence (TCE) to find if it can help detect equivalent mutants in C++ programs. We observed that compiler optimizations at assembly level code can indeed detect equivalent mutants and that different optimizations vary in detecting them. We conclude that -O3 optimization is generally more efficient to detect equivalent mutants. Moreover, -O2 and -Os optimizations perform similarly in detecting equivalent mutants. In future, we are interested to know which mutation operators contribute more/less in generating equivalent mutants and which optimization techniques detect the equivalent mutants.

REFERENCES

[1] T. Titcheu Chekam, M. Papadakis, T. F. Bissyande, Y. Le ´ Traon, and K. Sen, "Selecting fault revealing mutants," Empirical Software Engineering, vol. 25, no. 1, pp. 434– 487, 2020.

[2] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," IEEE Transactions on Software Engineering, vol. 37, no. 5, pp. 649–678, sep 2011.

[3] G. Petrovic and M. Ivankovi ´ c, "State of mutation testing at ´ google," in Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ser. ICSE-SEIP '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 163–171.

[4] M. Beller, C.-P. Wong, J. Bader, A. Scott, M. Machalica, S. Chandra, and E. Meijer, "What it would take to use mutation testing in industry—a study at facebook," in 2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), 2021, pp. 268–277.

[5] G. Petrovic, M. Ivankovi ´ c, G. Fraser, and R. Just, "Does ´ mutation testing improve testing practices?" in 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), 2021, pp. 910–921.

[6] D. Schuler and A. Zeller, "(un-)covering equivalent mutants," in 2010 Third International Conference on Software Testing, Verification and Validation, 2010, pp. 45–54

[7] T. A. Budd and D. Angluin, "Two notions of correctness and their relation to testing," Acta Inf., vol. 18, no. 1, p. 31–45, mar 1982. [Online]. Available: https://doi.org/10. 1007/BF00625279.

[8] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon, "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique," in 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 1, 2015, pp. 936–946.

[9] D. Baldwin and F. Sayward, Heuristics for Determining Equivalence of Program Mutations, ser. Department of Computer Science: Research report. Yale University, Department of Computer Science, 1979

[10] P. Delgado-Perez, I. Habli, S. Gregory, R. Alexander, ´ J. Clark, and I. Medina-Bulo, "Evaluation of mutation testing in a nuclear industry case study," IEEE Transactions on Reliability, vol. 67, no. 4, pp. 1406–1419, 2018

[11] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. Le Traon, and M. Harman, "Detecting trivial mutant equivalences via compiler optimisations," IEEE Transactions on Software Engineering, vol. 44, no. 4, pp. 308–333, 2018

[12] M. Houshmand and S. Paydar, "Tce+: An extension of the tce method for detecting equivalent mutants in java programs," in Fundamentals of Software Engineering, M. Dastani and M. Sirjani, Eds. Cham: Springer International Publishing, 2017, pp. 164–179.

[13] P. Delgado-Perez, I. Medina-Bulo, F. Palomo-Lozano, ´ A. Garc´ıa-Dom´ınguez, and J. J. Dom´ınguez-Jimenez, ´ "Assessment of class mutation operators for c++ with the mucpp mutation system," Information and Software Technology, vol. 81, pp. 169–184, 2017.

[14] H. Lin, Y. Wang, Y. Gong, and D. Jin, "Domain-rip analysis: A technique for analyzing mutation stubbornness," IEEE Access, vol. 7, pp. 4006–4023, 2019