Study of Trivial Compiler Equivalence on C++ Object-Oriented Mutation Operators

Pedro Delgado-Pérez
Escuela Superior Ingeniería, Universidad de Cádiz
Puerto Real, Spain
pedro.delgado@uca.es

Sergio Segura
ETS Ingeniería Informática, Universidad de Sevilla
Sevilla, Spain
sergiosegura@us.es

ABSTRACT

Trivial Compiler Equivalence (TCE) has been recently proposed as an effective technique to detect equivalences between programs, where two or more programs are equivalent if the compiler produces the same binary code. Mutation testing can greatly benefit from TCE as a way to reveal some equivalent and duplicate mutants, which traditionally hinder the applicability of the technique. For instance, previous research has shown that about 28% of the mutants generated by traditional mutation operators in C programs can be removed using TCE. However, the effectiveness of TCE has not been assessed with class-level operators, where the percentage of equivalent mutants is known to be higher than when using traditional ones. In this paper, we present an empirical study on the effectiveness of TCE at identifying equivalent and duplicate mutants using C++ class operators. The results show that TCE is helpful to discard equivalent and duplicate mutants: 241 out of 1,987 (12%) in our study, including 189 out of 684 (27.6%) manually-identified equivalent mutants. Large differences were observed among the different case studies, especially in the detection rate of equivalent mutants, which ranged from 4% to 45%.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**;

KEYWORDS

mutation testing;
equivalent mutants;
compiler optimizations;
object orientation;
C++.

1 INTRODUCTION

Mutation testing is a well-established technique for the assessment and improvement of test suites [20]. Given that the final goal of developing a test suite is to detect possible faults in our program, mutation testing focuses on analyzing the ability of the test cases to reveal plausible defects. To that end, faulty versions derived from the program under test are generated, called *mutants*. Such mutants are usually generated following a set of transformation rules known as *mutation operators*. If the mutants are *killed*, that is, if the original program and the mutants behave differently when executed against the test suite (there are observable differences in the outputs of these programs), we can be fairly confident on the fault detection ability of our test suite. Otherwise, undetected or *live mutants* may reveal weaknesses in the test cases.

We cannot say, however, that the possible enhancement in the test suite is proportional to the number of live mutants. This is because of the existence of *equivalent mutants*. An equivalent mutant has the same functionality as the original program. Therefore, it cannot be killed by any test case and it does not help to assess nor improve the test suite. This fact would not have further relevance if they could be easily discarded. However, identifying equivalent mutants is a costly task that places a significant burden on the tester. Despite being an undecidable problem, many researchers have proposed different approaches in the past to reduce its impact [6]. A recently proposed and promising strategy, especially because of its cost-benefit trade-off, leverages compiler optimizations to detect trivial equivalent mutants [19]. This technique, known as *TCE* or *Trivial Compiler Equivalence*, has proved useful to remove on average around 28% and 11% of all C and Java *method-level mutants* (also known as traditional mutants), respectively [12]. Moreover, the technique is appealing as it is affordable and can be easily and widely applicable.

Class or *object-oriented operators* were originally proposed by Kim et al. [11] to specifically target object-oriented features, such as inheritance or polymorphism. Currently, it is unknown the effectiveness of using compiler optimizations in the detection of equivalent class mutants. The given reason in previous research for excluding object-oriented operators [12] is that these operators produce a low number of mutants and equivalent mutants, based on the results of a previous paper [14]. While it is true that class operators engender a lower number of mutants than method-level operators, new studies have shown that the percentage of equivalent mutants generated with class operators is significantly higher. Namely, Segura et al. [22] found that 45.4% of the mutants were equivalent using class operators for Java, and Delgado-Pérez et al. [3] reported that 27.9% of class mutants for C++ were also equivalent. Additionally, Derezińska and Rudnik [5] identified 20.4% of C# class mutants as equivalent, even though they only reviewed a subset of the live mutants. In contrast, the percentage of equivalent mutants is between 5 and 15% for traditional operators [15]. For

instance, Segura et al. [22] identified 13.4% traditional mutants as equivalent in the same study.

Bearing in mind the difference in time and effort required to identify equivalent mutants manually and automatically through TCE, it is reasonable to apply TCE even if the number of mutants is not excessively high. Judging from the literature, method and class-level operators are the sets of operators most commonly used [20]. Taking into account the aforementioned two factors, there is a need to conduct new experiments to assess the effectiveness of TCE in relation to class operators and complete the study about the benefits of using TCE. In our study analyzing C++ test subjects, we find that a subset of *equivalent* and *duplicate* class mutants can also be detected using TCE. Remarkably, the results show that 12% of all mutants can be safely discarded, removing around 27% of the set of manually-identified equivalent mutants. TCE detected between 4.1% and 45% of these equivalent mutants in the test subjects. This suggests that the performance mostly depends on the features of the subject.

The paper is structured as follows. Section 2 describes TCE in further detail, comments the state of the art and explains how we apply the technique. Section 3 presents the experimental setup and Section 4 shows and discusses the results of the application of TCE to class mutants, including threats to validity. Finally, Section 6 presents the conclusions.

2 TRIVIAL COMPILER EQUIVALENCE

2.1 Definition

Trivial Compiler Equivalence is a technique that exploits the optimizations performed by existing compilers to detect equivalences in the set of mutants. Roughly speaking, when two programs are compiled enabling compiler optimizations, they can turn out to have the same *binary code* due to the transformations performed by the set of optimizations. By means of the comparison of those binary files, we can detect mutants that are *trivially equivalent* to the original program, and mutants that are trivially equivalent to other mutants, that is, *duplicate* mutants.

Formally, the application of TCE can be defined as follows. Let a and b be two programs, and Ω a binary relation such that $a\ \Omega\ b$ iff a and b have identical binary code. Also, let o be the original program and M the set of mutants generated from program o. Then:

- **TCE-equivalent mutants**: Let *TCE_EQ* be the set of equivalent mutants detected by TCE:

$$TCE\_EQ = \{a \in M \mid o\ \Omega\ a\} \tag{1}$$

- **TCE-duplicate mutants:** Let *TCE_DU* be the set of duplicate mutants detected by TCE:

$$TCE\_DU = \{a \in M \mid \exists b \in M : a\ \Omega\ b\} \tag{2}$$

These two properties hold:

$$\forall a, b \in TCE\_DU, a\ \Omega\ b \Rightarrow b\ \Omega\ a \tag{3}$$

$$\forall a, b, c \in TCE\_DU, a\ \Omega\ b \wedge a\ \Omega\ c \Rightarrow b\ \Omega\ c \tag{4}$$

Equation 3 and Equation 4 correspond to the symmetric and transitive relation respectively.

2.2 State of the art

Although compiler optimizations have been previously suggested to determine mutant equivalence [1], TCE was recently proposed by Papadakis et al. [19]. That study was later extended [12] by assessing Java in addition to C programs. In their studies, they found that approximately 28% and 11% of C and Java traditional mutants could be discarded respectively, and that around 30% and 50% of all equivalent mutants could be detected. The study by Delgado-Pérez [2] applying TCE to an industrial application supports TCE's effectiveness: around 35% of C traditional mutants could be removed considering equivalent and duplicate mutants.

The analysis of the efficiency [12, 19] revealed that TCE is reasonably fast (especially in equivalent mutant detection) in comparison to the effort of manual review as well as easily applicable. The application of TCE not only allows reducing the cost of mutation testing, but it also improves the accuracy of the mutation score, especially because duplicate mutants, which could inflate the metric, can be partially discarded now. For instance, previous experiments with C programs have shown that the use of this technique can improve the accuracy of the mutation score between 0% and 16% [12] and 10% on average [2].

Houshmand et al. [9] proposed an enhancement of TCE, which they called TCE+. In their study, they used an obfuscator for Java programs instead of the `javac` compiler, showing that the application of TCE in combination with an obfuscation tool can increase the effectiveness. TCE has also been applied to other sets of mutants, like memory mutants [24], revealing that about 5.5% of all mutants were TCE-equivalent or TCE-duplicate.

2.3 Application of TCE in our study

Figure 1 depicts a diagram which illustrates the procedure for equivalent mutant detection through compiler optimizations. In this study, we assess the set of class mutation operators implemented in the C++ mutation tool `MuCPP` [3]. As a first step, the mutation tool is applied to a C++ project to generate a set of class mutants. Then, both the original and the mutated versions are compiled with `g++` using the same set of optimizations, which results in binary files. `MuCPP` stores the mutants by means of the version control system `git` instead of as conventional files. As such, the binary files are compared using the option `diff` for binary files provided by `git`. For instance, mutant *m*1 can be compared with the original program (which is saved in the branch *master*) through the following command:

```
git diff --binary master m1 binaryfile
```

Regarding duplicate mutants, each mutant is compared with the rest of the mutants, which makes this process more expensive than detecting equivalent mutants. Nevertheless, the need for computation can be considerably reduced by only comparing the mutations contained in the same unit (e.g., mutants in the same class). It is unlikely that mutations injected into different classes turn out to be equivalents.

As shown by Houshmand et al. [9], we support the application of TCE via scripts instead of integrating the technique as a further functionality of a mutation tool. Thanks to this, we can reuse the

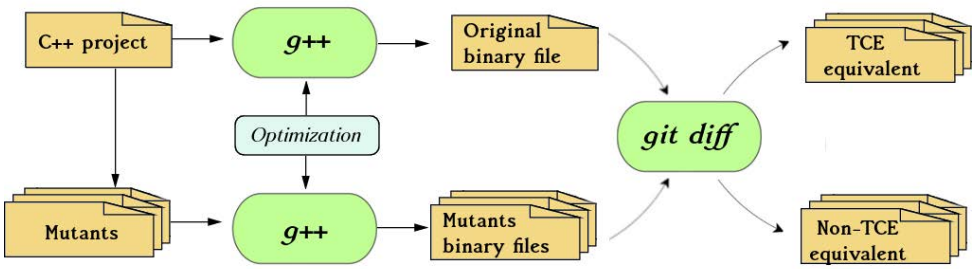

Figure 1: Diagram of the application of TCE to detect equivalent mutants in this work.

same scripts for different tools (provided that they follow the conventional mutation testing process) without the need to modify the source code of each tool.

3 EXPERIMENTAL SETUP

The aim of our experiments is to answer two main research questions regarding C++ class-level mutants:

> ***Research question 1:*** *What is the rate of equivalent and duplicate mutants detected by TCE when applied to class mutation operators? What is the impact of the different optimization options?*

> ***Research question 2:*** *What are the class operators that often generate TCE-equivalent and TCE-duplicate mutants?*

The experiments have been conducted on a computer running Ubuntu 14.04 LTS with gcc 4.8 (both the version of the operating system and the compiler are the same used in the study by Kintis et al. [12]). We have compiled the programs using four different levels of optimization[1]: `None` (default optimization option), `-O`, `-O2` and `-O3`. Additionally, we used `git` 1.9.1 to compare the binary files. The class operators that generated mutants in our study can be seen in Table 1 (further information about this set of operators can be found in a previous work [3]).

We have used a set of five C++ object-oriented systems, shown in Table 2. The mutants generated in these programs were manually reviewed to determine which of them were equivalent. As it can be seen in Table 2, the percentage of equivalent mutants (34.4%) is in line with the ratios of class equivalent mutants found in previous studies on object-oriented operators (see discussion in Section 1). That set of equivalent mutants is used as the ground truth in our experiments. We do not know, however, about the existing equivalences among mutants. As such, the detection capability of duplicate mutants is calculated taking as reference all mutants except those in the set *TCE_EQ* (i.e., TCE-equivalent mutants). Note that, because of the transitive relation (see Equation 4), if $o \ \Omega \ a$ and $o \ \Omega \ b$, then $a \ \Omega \ b$. Therefore, TCE-equivalent mutants have to be discarded to avoid that they are counted also as duplicate.

As mentioned earlier, this work is partially related to the work of Papadakis et al. [19], where the effectiveness of TCE was evaluated in the context of C traditional operators. Since we are applying the same approach as in their study (that is, we are applying the same compiler and the same optimizations levels), and there is no material difference between compiling traditional and class mutants in terms of performance, we do not replicate the study about the efficiency of TCE in this paper.

Table 1: Set of C++ object-oriented mutation operators examined in the study

Group	Op.	Description
Inheritance	**IHI**	Hiding variable insertion
	ISD	Base keyword deletion
	ISI	Base keyword insertion
	IOD	Overriding method deletion
	IOP	Overriding method calling position change
	IOR	Overriding method rename
	IPC	Explicit call of a parent's constructor deletion
Polymorphism and dynamic binding	**PCI**	Type cast operator insertion
	PMD	Member variable declaration with parent class type
	PPD	Parameter variable declaration with child class type
	PNC	New method call with child class type
Method overloading	**OMD**	Overloading method deletion
	OMR	Overloading method contents replace
	OAN	Argument number change
Object and member replacement	**MCO**	Member call from another object
	MCI	Member call from another inherited class
Miscellany	**CTD**	*this* keyword deletion
	CTI	*this* keyword insertion
	CID	Member variable initialization deletion
	CDC	Default constructor creation
	CDD	Destructor method deletion
	CCA	Copy constructor and assignment operator overloading deletion

[1]Information about these optimization options can be found here: https://gcc.gnu.org/onlinedocs/gcc-4.8.2/gcc/Optimize-Options.html

Table 2: Mutants in the C++ test subjects and number and proportion (% Eq.) of manually-identified equivalent mutants

Program	*Classes*	*Mutants*	*Equivalent*	*% Eq.*
Matrix TCL Pro	9	135	20	14.8
Dolphin	13	208	69	33.2
TinyXML2	20	433	91	21.0
MySQLServer	8	530	268	50.5
QtDOM	11	681	236	34.6
Total	61	**1,987**	**684**	**34.4**

Table 3: Number (No.) and proportion (%) of equivalent mutants detected by TCE in each program using the different optimization levels.

Prog.	**None**		**-O**		**-O2**		**-O3**	
	No.	%	No.	%	No.	%	No.	%
TCL	2	10	9	45	9	45	9	45
DPH	14	20.3	19	27.5	19	27.5	19	27.5
TXM	22	24.2	29	31.9	30	33	30	33
SQL	8	3	11	4.1	11	4.1	11	4.1
DOM	20	8.5	20	8.5	20	8.5	20	8.5
Total	66	9.6	88	12.9	89	13	89	13

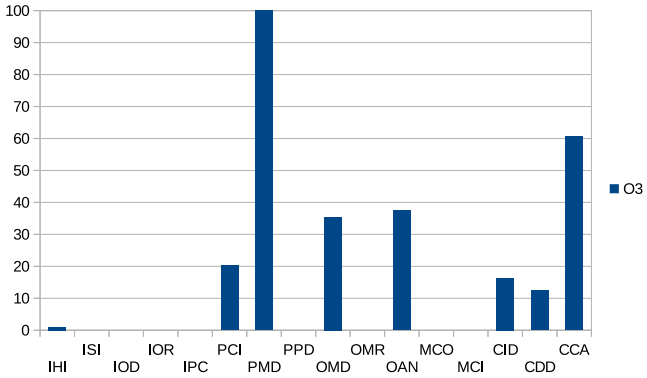

Figure 2: Percentage of TCE-equivalent mutants detected per mutation operator using -O3, taking into account all the analyzed programs. Only mutation operators that generated at least one equivalent mutant are considered.

4 RESULTS AND DISCUSSION

4.1 Equivalence detection

Table 3 shows the results of applying TCE to our test subjects with the four optimization levels. Namely, the number of equivalent mutants detected and the proportion with respect to the set of manually-identified equivalent mutants. We can highlight the following findings:

- TCE was able to detect up to 13% of all equivalent mutants identified in our programs (89 out of 684) with the highest optimization options (`-O2` and `-O3`). Looking at the results of the programs individually, we can observe a varying detection power, from 4.1% to 45%. This fact suggests that the effectiveness of TCE greatly depends on the features of the subject under test.
- Applying no optimizations (`None`) was the least effective option by far. Except for *DOM*, where the results for all optimization levels were the same, `None` is the option that detected fewer equivalent mutants, with significant differences in *TCL* or *TXM*. Besides, we should note also that all equivalent mutants revealed with `None` were also revealed by the other options.
- Regarding the rest of optimization levels, there was no difference between `-O2` and `-O3`, and no meaningful differences with `-O` (just one mutant generated by the operator *CID* was not detected by `-O` in comparison with the other more demanding options). Again, all the mutants detected by `-O` were detected with `-O2` and `-O3` as well.

In order to know about the distribution of TCE-equivalent mutants in the set of mutation operators, Table 4 furnishes the percentage of equivalent mutants detected per operator in each program and optimization level. Figure 2 helps to interpret the performance of TCE in each mutation operator globally. As it can be seen, TCE was able to identify equivalent mutants in 8 out of 16 mutation operators that generated at least one equivalent mutant. These 8 operators belong to different categories: inheritance (*IHI*), polymorphism (*PMD* and *PCI*), method overloading (*OAN* and *OMD*) and operators related to construction and destruction of objects (*CCA*, *CID* and *CDD*). The best operators in terms of TCE detection were *PMD*, *CCA*, *OAN* and *OMD*, with a ratio TCE-equivalent to manually-identified equivalent mutants over 30% (see Figure 2). It is remarkable that all equivalent mutants from *PMD* were detected; also, a subset of the equivalent mutants from *PCI* was revealed in all the programs in which this operator produced some equivalent mutants. In contrast, it is the fifth operator in which TCE seems to be more effective.

4.2 Duplicate mutant detection

Table 5 presents the results of the detection of duplicate mutants (number and proportion) thanks to TCE. The minimum and maximum percentage of duplicate mutants found was 1.1% and 18.5% respectively. The option `-O3` was the most effective, although all options performed similarly. We should note that the number of duplicate mutants shown in this table represents the number of mutants that are not necessary because there are other mutants with the same binary code. For instance, if three mutants are equivalent among them, that means that two of them can be discarded and one should be kept in the set. Recall that the percentages of TCE-equivalent and TCE-duplicate mutants (respectively shown in Table 3 and Table 5) should not be interpreted in the same way; we use manually-equivalent mutants as ground truth to calculate the proportion of TCE-equivalent mutants, while the detection rates of TCE-duplicate mutants take the complete set of mutants except those in *TCE_EQ* as reference point.

Table 4: Percentage of TCE-equivalent mutants divided by operator, case study and optimization level. Those optimization levels that reported the same percentages in each program have been grouped under a common heading (for instance, -O/-O3 means that the options O, -O2 and -O3 reported the same results).

Op.	TCL		DPH		TXY			SQL		DOM
	None	-O/-O3	None	-O/-O3	None	-O	-O2/-O3	None	-O/-O3	None/-O3
IHI					0	0	0	0	0	3.1
ISI			0	0				0	0	0
IOD			0	0	0	0	0	0	0	0
IOR			0	0	0	0	0			0
IPC			0	0				0	0	
PCI					32.7	38.5	38.5	12.5	12.5	9.5
PMD					100	100	100	100	100	100
PPD					0	0	0			0
OMD	25	62.5	0	100	0	14.3	14.3	0	0	0
OMR	0	0	0	0				0	0	
OAN								0	37.5	
MCO			0	0	0	0	0	0	0	0
MCI					0	0	0			
CID	0	0	72.2	72.2	0	0	10	0	0	0
CDD	0	50	0	0	0	0	0			0
CCA	0	42.9	20	100	50	100	100			0

Table 6 shows the number of duplicate mutants between mutation operators in a matrix-like structure. We should note that we did not find any case where more than two operators generated duplicate mutants among them. In other words, all mutants in *TCE_DU* (i.e., the set of TCE-duplicate mutants) were equivalent to other mutants generated by the same operator (as in the case of *IHI* and *MCO*) or equivalent to just another operator (e.g., *CTI* and *CID*). It is interesting to remark that the percentage of duplicate mutants between two operators (e.g., *IOD* and *OMD*) accounted for about 14% of the mutants in *TCE_DU*, and only six operators generated such mutants (*IOD*, *OMD*, *CTD*, *CTI*, *CID* and *CCA*). Nevertheless, these low numbers are not surprising: unlike traditional operators, it is known that each class mutation operator addresses a different object-oriented feature [4] and, as such, it is unlikely that many class mutants result in the same binary code.

Table 5: Number (No.) and proportion (%) of duplicate mutants detected by TCE in each program using the different optimization levels.

Prog.	None		-O		-O2		-O3	
	No.	%	No.	%	No.	%	No.	%
TCL	3	2.3	3	2.3	3	2.3	3	2.3
DPH	2	1.1	4	2.1	4	2.1	4	2.1
TXM	24	6	24	6	24	6	25	6.2
SQL	94	18.1	96	18.5	96	18.5	96	18.5
DOM	23	3.5	24	3.6	24	3.6	24	3.6
Total	146	7.6	151	7.8	151	7.8	152	7.9

Note that the symmetric relation (see Equation 3) implies that any mutant involved in an equivalence relation can be removed without prejudicing the assessment. By reviewing duplicate mutants, we found out that, under certain circumstances, the operators *IOD*, *OMD* and *CCA* address the same method. As such, we could work on the generation phase of these operators to prevent overlap.

Table 6: Matrix with the number of duplicate mutants between operators. Given the symmetric relation, the equivalence between operators is represented in both directions.

Op.	IHI	IOD	OMD	MCO	CTD	CTI	CID	CCA
IHI	124							
IOD			6					
OMD		6						12
MCO				7				
CTD						1		
CTI					1		2	
CID						2		
CCA			12					

Duplicate mutants from *IHI* mostly appear when the inserted variables are not referenced in the child classes. This situation should be explored further (for instance, taking into account friendship relations), to understand the conditions that lead to these duplicate mutants and whether they can be avoided.

4.3 Joint result

Table 7 summarizes the joint ratio of both TCE-equivalent and TCE-duplicate mutants to the full set of mutants. In this way, we can better observe the total number of invaluable mutants that can be removed.

At this point, we should note that, while a manually-identified equivalent mutant might not be detected as TCE-equivalent, it might be identified as TCE-duplicate with other equivalent mutants. This situation is graphically shown in Figure 3 with the shaded region. Therefore, we additionally analyzed how many of the TCE-duplicate mutants were in the set of manually-identified equivalent

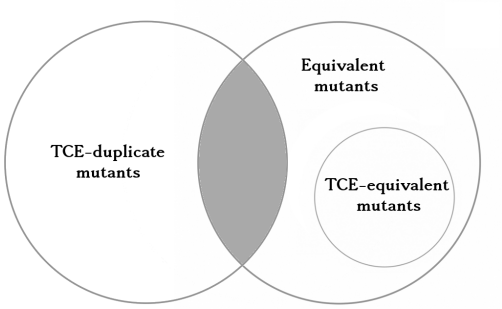

Figure 3: Venn diagram with the sets of manually-identified equivalent, TCE-equivalent and TCE-duplicate mutants.

mutants at the same time. Remarkably, 100 out of 152 TCE-duplicate mutants were known to be equivalent. Since these mutants can also be discarded, they have been added to the number of TCE-equivalent mutants shown in Table 3. Columns *Eq.* and *%Eq.* show the total number and the final percentage of equivalent mutants that can be removed using `-O3`. Altogether, this percentage increases from 13% to 27.6%.

Finally, we have observed a great variation in the ratio of detection of equivalent and duplicate mutants in the different programs (see Tables 3 and 5). These ratios do not seem to relate to the size of the set of mutants generated in the programs. This observation is in line with the correlation measures performed by Kintis et al. [12].

Table 7: Proportion of mutants detected by TCE (equivalent and duplicate mutants jointly) with the different options. Number (Eq.) and proportion (% Eq.) of equivalent mutants removed using -O3 (counting TCE-duplicate mutants as well) are also shown.

Prog.	None	-O	-O2	-O3	Eq.	% Eq.
TCL	3.7	8.9	8.9	8.9	10	50
DPH	7.7	11.1	11.1	11.1	19	27.5
TXM	10.6	12.2	12.4	12.7	37	40.7
SQL	19.2	20.2	20.2	20.2	99	36.9
DOM	6.3	6.5	6.5	6.5	24	10.2
Total	10.7	12	12	12.1	189	27.6

4.4 Answer to research questions

> ***Research question 1:*** *What is the rate of equivalent and duplicate mutants detected by TCE when applied to class mutation operators? What is the impact of the different optimization options?*

The percentage of mutants from the total that can be removed thanks to TCE is 12.1% (241 out of 1,987). This percentage is similar to the number of method-level mutants that could be discarded in Java (11%) [12], but lower than in C programs (28%). The percentage of equivalent mutants that can be removed is 27.6% (189 out of 684). However, the detection of TCE-equivalent mutants was quite different in the programs (4.1% and 45% in the worst and best case). Apparently, applying the highest optimization level is not decisive. Given that the higher the optimization option the more time it takes to compile mutants [12], using `-O` seems to be a suitable option to save in compilation time.

> ***Research question 2:*** *What are the class operators that often generate trivial equivalent and duplicate mutants?*

TCE detected equivalent mutants in half of the operators that generated some equivalent mutants in our programs. Among them, TCE performed better with equivalent mutants generated by *PMD* (100%), *CCA* (60%), *OAN* (37%) and *OMD* (35%) than those produced by *PCI* (20%), *CID* (16%), *CDD* (12%) and *IHI* (1%). There were also eight operators involved in the detection of duplicate mutants, although many of them were generated by *IHI*, and *CTI* and *CTD* only generated one TCE-duplicate mutant between them. Given the specificity of class operators, there are no many equivalence relations among them.

4.5 Threats to validity

There are two main threats to the validity of the presented results, which are common in mutation-based assessments:

- **Equivalence:** Being the sets of equivalent mutants reviewed in a manual way, we might have failed when classifying some of them. However, in that case, it is likely that the proportion of equivalent mutants detected is greater than the one reported in the paper (i.e., some of those mutants could be actually killable).
- **Generalization of the results:** Evaluations such as the one performed in these experiments are judged with respect to their representativeness. To counter this threat, we analyzed almost 2,000 mutants, of which 684 were equivalent. While the Yao et al. [25] benchmark used to test the effectiveness of TCE in C programs contains 990 equivalent mutants [12], our test subjects surpass the manually-analyzed Java mutants used by Kintis et al. [12] (1,542 mutants and 196 equivalent mutants) and by Houshmand et al. [9] (1,872 mutants and around 100 equivalent mutants).

There is a further threat, which is related to the underlying technologies used to apply TCE. We have not found false positives in our study but we cannot guarantee that the compiler is exempt from defects, such as the ones detected by Tao et al. [23] or the false positive reported by Delgado-Pérez et al. [2]. In any event, it is unlikely that such defects could greatly impact the results shown in the paper. The rest of the tools utilised (`git` and `MuCPP`) can also affect the results. In the case of `MuCPP`, this mutation tool implements some rules to avoid the generation of equivalent mutants through static analysis [3]. As such, the results might be different using other mutation tools with class operators.

5 RELATED WORK

The automatic detection of equivalent mutants to reduce the cost of mutation testing is a topic that has been widely studied in the past [16, 20]. Some works, such as the one by Grün et al. [7], have

studied mutation operators that generate equivalent mutants, and have tried to identify the causes for the generation of these mutants. As a result, equivalence conditions can be detected and integrated into mutation tools to avoid their generation [3, 10].

The review by Madeyski et al. [16] identified a set of relevant techniques to address the equivalent mutant problem. Among them, Offutt and Pan [18] devised a technique called constrained-based test data to generate input data that are useful to identify equivalent mutants. Hierons et al. [8] applied program slicing to help in the detection of these mutants. Later, Schuler and Zeller [21] analyzed the coverage impact of mutations to alleviate the effects of the equivalence. Recently, Kintis et al. [13] found that higher-order mutation can help to isolate equivalent mutants.

Other works are based on compiler optimizations to identify some equivalent mutants. The first heuristics for detecting equivalence were proposed by Baldwin and Sayward [1] and later studied by Offutt and Craft [17]. Recently, Papadakis et al. [19] proposed the technique called TCE to detect equivalent mutants by comparing the binary files of the original program and the mutants once the optimizations of the compiler have been applied. Their study was later extended by Kintis et al. [12], who studied the efficiency and effectiveness of TCE in Java programs in addition to C programs.

6 CONCLUSION

Mutation testing is a powerful technique that suffers from excessive cost. The attempts to reduce the expense, especially that stemming from equivalence, can favour a wider application of this technique by practitioners. However, the empirical studies with regard to TCE are still limited despite its promising results so far. The experiments in this paper show that TCE is also useful to reduce the cost when addressing object-oriented mutants, showing similar performance as in previous studies.

As future work, we should learn about the set of optimizations implemented in different compilers to improve TCE effectiveness, or even to devise new equivalence rules that could be integrated into mutation tools.

7 ACKNOWLEDGMENTS

This work has been partially supported by the European Commission (FEDER) and Spanish Government under MINECO projects DArDOS (TIN2015-65845-C3-3-R) and BELI (TIN2015-70560-R).

REFERENCES

[1] D. Baldwin and F. Sayward. 1979. *Heuristics for Determining Equivalence of Program Mutations.* Yale University, Department of Computer Science.

[2] Pedro Delgado-Pérez, Ibrahim Habli, Steve Gregory, Rob Alexander, John Clark, and Inmaculada Medina-Bulo. 2018. Evaluation of Mutation Testing in a Nuclear Industry Case Study. *IEEE Transactions on Reliability* 99 (2018), 1–14. https://doi.org/10.1109/TR.2018.2864678

[3] Pedro Delgado-Pérez, Inmaculada Medina-Bulo, Francisco Palomo-Lozano, Antonio García-Domínguez, and Juan José Domínguez-Jiménez. 2017. Assessment of class mutation operators for C++ with the MuCPP mutation system. *Information and Software Technology* 81 (2017), 169–184. https://doi.org/10.1016/j.infsof.2016.07.002

[4] Pedro Delgado-Pérez, Sergio Segura, and Inmaculada Medina-Bulo. 2017. Assessment of C++ object-oriented mutation operators: A selective mutation approach. *Software Testing, Verification and Reliability* 27, 4-5 (2017). https://doi.org/10.1002/stvr.1630

[5] Anna Derezińska and Marcin Rudnik. 2012. Quality Evaluation of Object-Oriented and Standard Mutation Operators Applied to C# Programs. In *Objects, Models, Components, Patterns*, Carlo A. Furia and Sebastian Nanz (Eds.). Lecture Notes in Computer Science, Vol. 7304. Springer Berlin Heidelberg, 42–57. https://doi.org/10.1007/978-3-642-30561-0_5

[6] F. Cutigi Ferrari, A. Viola Pizzoleto, and J. Offutt. 2018. A Systematic Review of Cost Reduction Techniques for Mutation Testing: Preliminary Results. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 1–10. https://doi.org/10.1109/ICSTW.2018.00021

[7] Bernhard J. M. Grün, David Schuler, and Andreas Zeller. 2009. The Impact of Equivalent Mutants. In *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops (ICSTW '09)*. IEEE Computer Society, Washington, DC, USA, 192–199. https://doi.org/10.1109/ICSTW.2009.37

[8] Rob Hierons, Mark Harman, and Sebastian Danicic. 1999. Using program slicing to assist in the detection of equivalent mutants. *Software Testing, Verification and Reliability* 9, 4 (1999), 233–262. https://doi.org/10.1002/(SICI)1099-1689(199912)9:4<233::AID-STVR191>3.0.CO;2-3

[9] Mahdi Houshmand and Samad Paydar. 2017. TCE+: An Extension of the TCE Method for Detecting Equivalent Mutants in Java Programs. In *Fundamentals of Software Engineering*, Mehdi Dastani and Marjan Sirjani (Eds.). Springer International Publishing, Cham, 164–179. https://doi.org/10.1007/978-3-319-68972-2_11

[10] J. Hu, N. Li, and J. Offutt. 2011. An Analysis of OO Mutation Operators. In *IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2011*. 334–341. https://doi.org/10.1109/ICSTW.2011.47

[11] Sunwoo Kim, J A Clark, and J A McDermid. 1999. Assessing Test Set Adequacy for Object-Oriented Programs using Class Mutation. In *Proceedings of the f the 3rd Symposium on Software Technology (SoST'99), Buenos Aires, Argentina*. 72–83.

[12] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. Le Traon, and M. Harman. 2018. Detecting Trivial Mutant Equivalences via Compiler Optimisations. *IEEE Transactions on Software Engineering* 44, 4 (April 2018), 308–333. https://doi.org/10.1109/TSE.2017.2684805

[13] Marinos Kintis, Mike Papadakis, and Nicos Malevris. 2015. Employing Second-order Mutation for Isolating First-order Equivalent Mutants. *Software Testing, Verification and Reliability* 25, 5-7 (Aug. 2015), 508–535. https://doi.org/10.1002/stvr.1529

[14] Yu-Seung Ma, Mary Jean Harrold, and Yong-Rae Kwon. 2006. Evaluation of Mutation Testing for Object-oriented Programs. In *Proceedings of the 28th International Conference on Software Engineering (ICSE '06)*. ACM, New York, NY, USA, 869–872. https://doi.org/10.1145/1134285.1134437

[15] Yu-Seung Ma, Yong Rae Kwon, and Sang-Woon Kim. 2009. Statistical Investigation on Class Mutation Operators. *ETRI Journal* 31, 2 (April 2009), 140–150. https://doi.org/10.4218/etrij.09.0108.0356

[16] Lech Madeyski, Wojciech Orzeszyna, Richard Torkar, and Mariusz Jozala. 2014. Overcoming the Equivalent Mutant Problem: A Systematic Literature Review and a Comparative Experiment of Second Order Mutation. *IEEE Transactions on Software Engineering* 40, 1 (Jan. 2014), 23–42. https://doi.org/10.1109/TSE.2013.44

[17] A. Jefferson Offutt and W. Michael Craft. 1994. Using compiler optimization techniques to detect equivalent mutants. *Software Testing, Verification and Reliability* 4, 3 (1994), 131–154. https://doi.org/10.1002/stvr.4370040303

[18] A. J. Offutt and Jie Pan. 1996. Detecting equivalent mutants and the feasible path problem. In *Proceedings of the Eleventh Annual Conference on Computer Assurance, 1996. COMPASS '96, Systems Integrity. Software Safety. Process Security.* 224 –236. https://doi.org/10.1109/CMPASS.1996.507890

[19] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. 2015. Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1 (ICSE'15)*. IEEE Press, Piscataway, NJ, USA, 936–946. https://doi.org/10.1109/ICSE.2015.103

[20] Mike Papadakis, Marinos Kintis, Jie Zhang, Yves Le Traon, and Mark Harman. 2018. Mutation Testing Advances: An Analysis and Survey. *Advances in Computers* (2018).

[21] David Schuler and Andreas Zeller. 2013. Covering and Uncovering Equivalent Mutants. *Software Testing, Verification and Reliability* 23, 5 (2013), 353–374. https://doi.org/10.1002/stvr.1473

[22] Sergio Segura, Robert M. Hierons, David Benavides, and Antonio Ruiz-Cortés. 2011. Mutation testing on an object-oriented framework: An experience report. *Information and Software Technology* 53, 10 (2011), 1124–1136. https://doi.org/10.1016/j.infsof.2011.03.006 Special Section on Mutation Testing.

[23] Q. Tao, W. Wu, C. Zhao, and W. Shen. 2010. An Automatic Testing Approach for Compiler Based on Metamorphic Testing Technique. In *2010 Asia Pacific Software Engineering Conference*. 270–279. https://doi.org/10.1109/APSEC.2010.39

[24] Fan Wu, Jay Nanavati, Mark Harman, Yue Jia, and Jens Krinke. 2017. Memory mutation testing. *Information and Software Technology* 81 (2017), 97 – 111. https://doi.org/10.1016/j.infsof.2016.03.002

[25] Xiangjuan Yao, Mark Harman, and Yue Jia. 2014. A Study of Equivalent and Stubborn Mutation Operators Using Human Analysis of Equivalence. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 919–930. https://doi.org/10.1145/2568225.2568265