Detecting Trivial Mutant Equivalences via Compiler Optimisations

Marinos Kintis, *Member, IEEE*, Mike Papadakis, *Member, IEEE*, Yue Jia, *Member, IEEE*, Nicos Malevris, Yves Le Traon, *Member, IEEE*, and Mark Harman, *Member, IEEE*.

Abstract—Mutation testing realises the idea of fault-based testing, i.e., using artificial defects to guide the testing process. It is used to evaluate the adequacy of test suites and to guide test case generation. It is a potentially powerful form of testing, but it is well-known that its effectiveness is inhibited by the presence of equivalent mutants. We recently studied Trivial Compiler Equivalence (TCE) as a simple, fast and readily applicable technique for identifying equivalent mutants for C programs. In the present work, we augment our findings with further results for the Java programming language. TCE can remove a large portion of all mutants because they are determined to be either equivalent or duplicates of other mutants. In particular, TCE equivalent mutants account for 7.4% and 5.7% of all C and Java mutants, while duplicated mutants account for a further 21% of all C mutants and 5.4% Java mutants, on average. With respect to a benchmark ground truth suite (of known equivalent mutants), approximately 30% (for C) and 54% (for Java) are TCE equivalent. It is unsurprising that results differ between languages, since mutation characteristics are language-dependent. In the case of Java, our new results suggest that TCE may be particularly effective, finding almost half of all equivalent mutants.

Index Terms—Mutation Testing, Equivalent Mutants, Duplicated Mutants, Compiler Optimisation.

1 Introduction

Mutation testing [1], [2] has attracted a lot of interest, because there is evidence that it is capable of simulating real faults [3], [4], [5] and subsuming other popular test adequacy criteria [6], [7], [8], [9]. It can also be used as a technique for generating test data [10], [11], as well as for assessing test data quality and can also explore subtle faults [12], [13] in the presence of fault masking and failed error propagation [14].

A mutant is a syntactically altered version of the program under test. The syntactic alterations are typically small, and are designed to reflect typical faults that might reside in the original program. A mutant is said to be killed, if a test case can be found that distinguishes between the mutant and the original program. The underlying idea of mutation testing is that test suites that kill many mutants will tend to be of higher quality than those that kill fewer. In this way, mutation testing can be used to assess the quality of a test suite, and can also be used to help the test case generation, by guiding the construction of test cases towards those that kill mutants.

However, at the heart of mutation testing lies a problem that has been known to be undecidable for more than three decades [15]: the equivalent mutant problem. That is, mutation testing might produce a mutant that is syntactically different from the original, yet semantically identical. In general, determining whether a syntactic change yields a semantic difference is undecidable. As a result, the tester would never know whether he or she has failed to find a killing test case because the mutant is particularly hard to kill, yet remains killable (a 'stubborn' mutant [16]), or whether failure to find a killing test case derives from the fact that the mutant is equivalent.

A related, newly identified problem, is the problem of mutant duplication. A duplicated mutant is simply a mutant that is semantically equivalent to some other mutant, although both duplicated mutants maybe semantically different from the original program. Duplicated mutants are also a problem for mutation testing, because they may artificially inflate the apparent mutant killing power of a test suite; a test case that kills two or more duplicated mutants is, all else being equal, no better than another test case that kills only a single non-duplicated mutant.

Techniques such as mutant sampling [17], [18], [19], higher order mutation [20], [21], [22], and mutant execution optimisation [23], [24], [25], can be used to reduce the number of mutants that need to be considered, but not necessarily the proportion that remain equivalent, nor the proportion of those that are duplicated.

Although theoretically undecidable, practical techniques may be able to significantly dent the equivalent and duplicate problems by detecting a proportion of equivalent/duplicated mutants. Equivalent mutant detection techniques have been extensively studied since 1979. Nevertheless, until now, no scalable, widely applicable technique has yet been found. Previous work on the detection of equivalent mutants has involved complicated program transformation techniques, which have proved difficult to scale and, thereby, have remained insufficiently practical to find implementation in current mutation testing tools

- *M. Kintis, M. Papadakis and Y. Le Traon are with the Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg.*
  E-mail: marinos.kintis@uni.lu, michail.papadakis@uni.lu and yves.letraon@uni.lu
- *Yue Jia and Mark Harman are with the CREST Centre, University College London, UK.*
  E-mail: yue.jia@ucl.ac.uk and mark.harman@ucl.ac.uk
- *N. Malevris is with the Athens University of Economics and Business, Greece.*
  E-mail: ngm@aueb.gr

and techniques. The equivalent mutant problem therefore remains the single most potent barrier to the wider uptake and exploitation of the potential power of mutation testing.

In this paper we study Trivial Compiler Equivalence (TCE) as a simple, fast and widely applicable technique for detecting equivalent mutants. The paper is an extension of our previous ICSE conference paper [26], which studied the application of TCE to the detection of equivalent and duplicated mutants in the C programming language. The present paper extends this previous study to also consider the Java programming language, allowing us to compare TCE performance on these two widely-used languages. The extended results further confirm that TCE is a highly effective and readily applicable technique, with strong evidence to suggest that it may be even more effective when applied to Java than what is already known to be when applied to C.

Specifically, while TCE finds, on average, approximately one third of the equivalent mutants in C programs, it finds approximately half of the equivalent mutants in Java. These new findings for Java are based on the study of a known equivalent ground truth set, which we have augmented for this study (and make available for replication and further study[1]). We also study the application of TCE to much larger Java programs, for which no ground truth is available, reporting results for the total number of equivalent and duplicated mutants found (using both the standard Java compiler[2] and the SOOT analysis framework[3]).

Overall, we believe that the findings regarding TCE are extremely encouraging. It can ameliorate the adverse effects of the equivalent and duplicated mutant problems for both C and Java programs by removing such invaluable mutants (by an average of approximately 10% for Java and nearly 30% for C) and, as a consequence, reduces the overall work needed to develop mutation adequate test suites by approximately 37%, while, at the same time, improving the accuracy of the mutation score measurement by 0%-18% for Java and 0%-16% for C (depending on the ratio of the killed mutants). Furthermore, and fundamental to its success and importance, TCE is not a complicated technique; it can easily be implemented and added to any mutation testing study. It has already been included in the mutation testing tool MILU (Version 3.2), and we were easily able to incorporate TCE analysis into the results produced by the C and Java mutation testing tools PROTEUM [27] and MUJAVA [28].

The rest of the paper is organised as follows: Section 2 presents mutation testing and related approaches. Section 3 details our experiment and the studied research questions, while, Sections 4 and 5 analyse our results. Our findings are discussed in Section 6. Finally, the threats to validity are presented in Sections 7, while Section 8 concludes with potential directions for future work.

1. http://pages.cs.aueb.gr/~kintism/papers/tce/ and http://www0.cs.ucl.ac.uk/staff/Y.Jia/projects/compiler_equivalence/
2. http://www.oracle.com/technetwork/java/index.html
3. http://sable.github.io/soot/

2 BACKGROUND

2.1 Mutation Testing

Mutation testing embeds artificial defects on the programs under test. These defects are called *mutants* and they are produced by simple syntactic rules, e.g., changing a relational operator from $>$ to $\geq$. These rules are called *mutant operators*. By applying an operator only once, i.e., the defective program has only one syntactic difference from the original one, a mutant called a *first order* mutant is produced. By making several syntactic changes i.e., applying the operators multiple times, a *higher order* mutant is produced. In this paper we consider only first order mutants. These are generated by applying the operators at all possible locations of the program under test, as supported by the 3.2 version of MILU and version 3 of MUJAVA. Additional information about the corresponding operators can be found at Section 3.4.

By measuring the ability of the test cases to expose mutants, an effectiveness measure can be established. Mutants are exposed when their outputs differ from those of the original program. When a mutant is exposed, it is termed *killed*, while in the opposite case, *live*. Of course, ideally, equivalent mutants should be removed from the test effectiveness assessment. Doing so gives the effectiveness measure called *mutation score*, i.e., the ratio of the exposed mutants to the number of the introduced, excluding the equivalent ones.

2.2 Equivalent Mutants

Early research on mutation testing has demonstrated that deciding whether a mutant is equivalent is an undecidable problem [15]. Undecidability of equivalences means that it is unrealistic to expect all the equivalent mutants to be removed; the best we can have here is just effective algorithms that can remove most equivalent mutants. Currently, a large number of mutants must pass a manual equivalence inspection [16]. This constitutes a significant cost. In addition, effort is wasted when testers generate test cases, either manually or automatically, in attempting to kill equivalent mutants. Apart from the human effort, there is a computational cost: since equivalent mutants cannot be killed, they have to be exercised on the entire test suite, whereas killable mutants only require the executions until they are killed.

Fortunately, partial and heuristic solutions exist [31]. However, tackling the equivalent mutant problem is hard. This is evident by the fact that very few attempts exist. According to a recent systematic literature review on the equivalent mutant problem [44], which identified 17 relevant techniques (in 22 articles), the problem is tackled in three ways. One is to address the problem directly by detecting some equivalent mutants, while, the other two try to reduce their effects by avoiding their creation of by suggesting likely non-equivalent ones to help with the manual analysis process. Following the terminology of Madeyski *et al.* [44], we refer to them as the *Detect*, *Avoid* and *Suggest* approaches, respectively.

Table 1 summarises the current state-of-the-art techniques in chronological order by focussing on the most recent techniques. Specifically, it records: the publication

TABLE 1: Summary of the related work on equivalent mutants.

Author(s) [Reference]	Year	Language	Largest Subject	#Eq. Mutants	Publicly Av. Tool	Category	Findings
Baldwin & Sayward [29]	1979	-	-	-	-	**Detect**	Compiler optimisation can detect eq. mutants
Acree [30]	1980	Fortran	-	25	-	**Detect**	Humans make mistakes when they identify eq. mutants
Offutt & Craft [31]	1994	Fortran	52	255	-	**Detect**	Compiler optimisation can detect eq. mutants
Offutt & Pan [32], [33]	1996-7	Fortran	29	695	✓	**Detect**	Constraint-based testing can detect eq. mutants
Voas & McGraw [34]	1997	-	-	-	-	**Detect**	Slicing may be helpful in detecting eq. mutants
Hierons *et al.* [35]	1999	-	-	-	-	**Detect** /Suggest	Program slicing can be used to detect and assist the identification of eq. mutants
Harman *et al.* [36]	2001	-	-	-	-	**Detect** /Suggest	Dependence analysis can be used to detect and assist the identification of eq. mutants
Adamopoulos *et al.* [37]	2004	-	-	-	-	Avoid	Co-evolution can help in reducing the effects of eq. mutants
Grun *et al.* [38]	2009	Java	12,449	8	✓	Suggest	Coverage Impact can be used to classify killable mutants
Schuler *et al.* [39]	2009	Java	94,902	10	✓	Suggest	Invariants violations can be used to classify killable mutants
Schuler & Zeller [40], [41]	2010-2	Java	94,902	63	✓	Suggest	Coverage Impact can be used to classify killable mutants
Nica & Wotawa [42]	2012	Java	380	1424	-	**Detect**	Constraint-based testing can detect eq. mutants
Kintis & Malevris [43]	2013	Java	25,909	84	-	Suggest	Mutants belonging to software clones exhibit analogous behaviour with respect to their equivalence
Madeyski *et al.* [44]	2014	Java	80,023	207	-	Avoid	Second order equivalent mutants are significantly less than the first order ones.
Kintis *et al.* [45], [46]	2012-4	Java	94,902	89	-	Suggest	Higher order mutants can be used to classify killable mutants
Papadakis *et al.* [47]	2014	C	513	5,589	-	Suggest	Coverage Impact can be used to classify killable mutants
Kintis & Malevris [48], [49]	2014-5	Java	25,909	165	-	**Detect**	Data-flow patterns can be used to detect eq. and partially eq. mutants
Bardin *et al.* [50]	2015	C	319	118	✓	**Detect**	Static analysis techniques, such as Value Analysis and Weakest Precondition calculus can identify eq. mutants
Papadakis *et al.* [26]	2015	C	362,769	9,551	✓	**Detect**	Compiler optimisations can be used to effectively automate the eq. mutant and duplicated mutant detection
This paper	**-**	**C - Java**	**362,769**	**13,455**	✓	**Detect**	Compiler optimisations can be used to effectively automate the eq. mutant and duplicated mutant detection

details, column "Author(s) [Reference]", the year of the publication, column "Year", the studied programming language, column "Language", the size of the largest program used, column "Largest Subject", the number of equivalent mutants studied, column "#Eq. Mutants", the existence of an automated publicly available tool, column "Publicly Av. Tool", the category of the approach, i.e., detection, avoidance or suggestion, column "Category" and the main findings of the publication, column "Findings". From this table it becomes evident that very few methods and tools exist. Regarding the equivalent mutant detection, only two publicly available tools exist with the largest considered subject being composed of 319 lines of code. It is noted that all the "large" subjects, i.e., having more than 1,000 lines of code, that were used in the previous research, involve a form of sampling. Mutants are sampled from the studied projects with no information about the relevant size of the component/class that these mutants are located. In these lines, in Table 1 we report the size of the projects that we consider. It is noted that the purpose of this table is to summarise the related work on equivalent mutants by focussing on the most recent advances. Further details on the subject can be found on the systematic literature review of Madeyski *et al.* [44].

Acree [30] studied killable and equivalent mutants, and found that testers correctly identified equivalent mutants for approximately 80% of the cases. In 12% of the cases, equivalent mutants were identified as killable and in 8%, killable mutants were identified as equivalent. Therefore, indicating that detection techniques, such as the one suggested by the present paper, not only help in saving resources but also at reducing the mistakes made by the humans.

The idea of using compiler optimisation techniques to detect equivalent mutants was suggested by Baldwin and Sayward [29]. The main intuition behind this technique is that code optimisation rules, such as those implemented by compilers, form transformations on equivalent programs. Thus, when the original program can be transformed by an optimisation rule to one of its mutants, then, this mutant is, *ipso facto*, equivalent. Baldwin and Sayward proposed adapting 6 compiler optimisation transformations. These transformations were then studied by Offutt and Craft [31] who implemented them inside Mothra, a mutation testing tool for Fortran. They found that on average 45% of the equivalent mutants can be detected. Our approach is inspired by this recruitment of compilers research to assist in equivalent mutant detection. As already discussed and demonstrated in the prior, conference version of this work [26], it is surprisingly effective for the case of the C programming language. However, we propose a truly simple (and therefore scalable and directly exploitable) use of compilers, which remained unexplored. Our TCE instead of deliberately implementing specialised techniques, it simply declares equivalences only for those mutants which their compiled object code is identical to the compiled object code of the original program. As indicated by our empirical findings, in Section 6, our approach is impressively effective, practical and scalable.

Offutt and Pan [32], [33] developed an automatic technique to detect equivalent mutants based on constraint solving. This technique uses mathematical constraints to formulate the killing conditions of the mutants. If these conditions are infeasible then, the mutants are equivalent.

Nica and Wotawa [42] implemented a similar constraint-based approach to detect equivalent mutants and demonstrated that many equivalent mutants can be detected. Voas and McGraw [34] suggested that program slicing can help in detecting equivalent mutants. Later, Hierons *et al.* [35] showed that amorphous program slicing can be used to detect equivalent mutants as well. Although potentially powerful, these techniques suffer from the inherent limitations of the constraint-based and slicing-based techniques.

It is evident that the constraint-based approach, [32], [33], was assessed on programs consisting of 29 lines of code at maximum, while, the slicing technique remains unevaluated apart from worked examples. The scalability of these approaches is inherently constrained by the scalability of the underlying constraint handling and slicing technology. Furthermore, a new implementation is required for every programming language to be considered. By contrast TCE applies to any language for which a compiler exists and so is as scalable as the compiler itself.

Kintis and Malevris [48], [49] used data-flow patterns and showed that a large proportion of equivalent mutants and partially equivalent mutants, i.e., mutants equivalent only under specific program paths, form data-flow anomalies. Bardin *et al.* [50] used static analysis techniques, such as Value Analysis and Weakest Precondition calculus, to detect mutants that are equivalent because they cannot be infected. Their results show that a significant number of those mutants can be detected. Although promising, these two methods have only been evaluated with less than 200 equivalent mutant instances and so their effectiveness, efficiency and practicality remain unknown.

Hierons *et al.* [35] suggested using program slicing to reduce the size of the program considered during the equivalence identification. Thus, testers can focus on the code relevant to the examined mutants. Harman *et al.* [36] also suggested using dependence-based analysis as a complementary method to assist in the detection of equivalent mutants.

Adamopoulos *et al.* [37] suggested the use of co-evolutionary techniques to avoid the creation of equivalent mutants. In this approach test cases and mutants are simultaneously evolved with the aim of producing both high quality test cases and mutants. However, these previous approaches have been evaluated only on case studies and synthetic data so their effectiveness and efficiency remains unknown.

More recently, several studies sought to measure the impact of mutant execution. Instead of finding a partial but exact solution to the problem, as done by the Detect approaches, they try to classify the mutants to help identify likely killable ones and likely equivalent ones, based on their dynamic behavior.

This idea was initially suggested by Grun *et al.* [38] and developed by the studies of Schuler *et al.* [39] and Schuler and Zeller [40], [41] who found that impact on coverage can accurately classify killable mutants. Kintis *et al.* [45], [46] further develop the approach, using the impact of mutants on other mutants, i.e., using higher order mutants. Papadakis *et al.* [47] proposed a mutation testing strategy that takes advantage of mutant classification. Finally, mutants belonging to software clones have been shown to exhibit analogous behaviour with respect to their equivalence [43]. Thus, knowledge about the (non-)equivalence of a portion of such mutants can be leveraged to analogously classify other mutants belonging to the same clones.

Apart from the technical differences between TCE and the existing approaches, as discussed above, there is also a fundamental difference that is the identification of duplicated mutants. Existing approaches only aim at equivalent mutants while TCE tackles the general problem of mutant equivalences.

2.3 Reducing the Cost of Mutation Testing

Mutant sampling has been suggested as a possible way to reduce the number of mutants. Empirical results demonstrate that even small samples [18] can be used as cost effective alternatives to perform mutation testing [17], [19]. Other approaches select mutant operators. Instead of sampling mutants at random, they select mutant operators that are empirically found to be the most effective. To this end, Offutt *et al.* [51] demonstrated that five mutant operators are almost as effective as the whole set of operators.

More recently, Namin *et al.* [52] used statistically identified optimal operator subsets. Other cost reduction methods involve mutant schemata [23], [53]. This technique works by parameterizing all the mutants through instrumentation, i.e., introduce all the mutants into one parameterised program. However, apart from the inherent limitations of this technique [28] and the execution overheads that introduces, it also makes all the equivalent mutant detection techniques not applicable.

Other approaches identify redundant mutants that fail to contribute to the testing process. Kintis *et al.* [54] defined the notion of disjoint mutants, i.e., a set of mutants that is representative of all the others (killing them implies killing all the others), and found that 9% of all mutants are disjoint. Ammann *et al.* defined minimal mutants using the notion of subsumption [55] and demonstrated that a small set of mutants, approximately 1.2% subsumes all the others. Based on these works, Papadakis *et al.* [56] demonstrated that redundancy among mutants has a very good chance ($> 60\%$) to inflate mutation score and lead to biassed results. Along the same lines, Kurtz *et al.* [57] analysed the validity of selective mutation and found that selective mutants score relatively low with respect to subsuming mutants.

Kaminski *et al.* [58], [59] and Just et al. [60] leverage the suggestions made by Tai [61] on fault-based predicate testing and demonstrated it possible to reduce the redundancy within the relational and logical operators. Higher order mutation can also reduce mutant numbers: Sampling [19], [44] and searching [13], [62], [63] within the space of higher order mutants both reduce the number of mutants and also of the equivalent mutants.

3 Experimental Study and Settings

This section details the settings of our experiment. First, it presents the TCE approach (Section 3.1) and the posed research questions (Section 3.2). Next, the studied C and Java programs are described (Section 3.3), along with the employed mutant operators (Section 3.4), and, finally, the execution environment (Section 3.5).

3.1 Detecting Mutant Equivalences: the TCE approach

Executable program generation involves several transformation phases that change the machine code. Different optimisation transformation techniques result in different executables. However, when there exist multiple program versions with identical source code, then there is no point in differentiating them with test data; it is safe to declare them as functionally equivalent. TCE realises this idea to detect mutant equivalences. It declares equivalent any two program versions with identical machine code. TCE simply compiles each mutant, comparing its machine code with that of the original program. Similarly, TCE also detects duplicated mutants, by comparing each mutant with the others residing in the same unit, i.e., function. As the reader will easily appreciate, the TCE implementation is truly trivial, hence its name: it is a compile command combined with a comparison of binaries.

3.2 Research Questions

The mutation testing process is affected by the distorting effects of the equivalent and duplicated mutants on the mutation score calculation. Therefore, a natural question to ask is how effective is the TCE approach in detecting equivalent and duplicated mutants. This poses our first RQ:

RQ1 (Effectiveness): How effective is the TCE approach in detecting equivalent and duplicated mutants?

We answer this question by reporting the prevalence of the equivalent and duplicated mutants detected by the TCE approach using `gcc`[4] and SOOT[5].

To reduce the confounding effects of different compiler configurations, we apply four and two popular options for `gcc` and SOOT on the selected classes/packages, and report the number of the equivalent and duplicated mutants found. SOOT does not support multiple levels of optimizations, thus, we only report its intra-procecural optimisations and report the equivalent and duplicated mutants found. The answer to this question also allows the estimation of the amount of effort that can be saved by the TCE method.

The existing mutant equivalent detection techniques suffer from performance and scalability issues. As a result, the authors are unaware of any mutation testing system that includes a proposed equivalent mutant detection. By contrast, the TCE is static, and can be applied to any program that can be handled by a compiler. This makes TCE potentially scalable, but we need additional empirical evidence to determine the degree to which it scales. Hence, in the second RQ, we seek to investigate the observed efficiency and the scalability of the TCE approach:

RQ2 (Efficiency): How efficient and scalable is the TCE approach?

To demonstrate the scalability, we use selected classes/packages from 12 large open source projects, 6 for each studied programming language, and we report the efficiency of the mutant generation, equivalent mutant detection and duplicated mutant detection processes. For the case of `gcc` we also explore the trade-off between the effectiveness and efficiency using different compiler settings.

To decide when it is appropriate to stop the testing process, testers need to know the mutation score. To this end, they need to identify equivalent mutants. The TCE approach improves the approximation by determining such mutants. However, to what extent? This is investigated in the next RQ:

RQ3 (Equivalent Mutants) What proportion of the equivalent mutants can be detected? What types of equivalent mutants can be detected?

To answer RQ3, we need to know the 'ground truth': how many equivalent mutants are there in the subjects studied? We therefore applied the TCE approach on two benchmark sets, one for each studied programming language, with hand-analysed, ground-truth data on equivalent mutants. The first benchmark[6], pertaining to the C test subjects, includes 990 manually-identified equivalent mutants over 18 small- and medium-sized subjects. The second one [7], for the Java programs, comprises 196 equivalent mutants selected over 6 small- and medium-sized subjects, detected with manual analysis.

We report the proportion of the equivalent mutants found by TCE. We also analyse and report the types of the detected equivalent mutants. This information is useful in the design of complementary equivalent detection techniques.

Mutation testers usually employ subsets of mutant operators. Therefore, knowing about the relationship between the operators and the equivalent and duplicated mutants found by TCE is useful in the sense that mutation testers can better understand the importance of their choices. Hence, the next RQ examines the extent of the equivalent and duplicated mutants found per mutant operator:

RQ4 (Impact on Mutant operators): What is the contribution of each operator to the proportion of equivalent and duplicated mutants found by TCE?

Among the several factors that can affect TCE is the program size. Thus, one might expect that in larger programs, the equivalent mutant identification would be

4. https://gcc.gnu.org/
5. http://sable.github.io/soot/
6. www0.cs.ucl.ac.uk/staff/Y.Jia/projects/equivalent_mutants/
7. http://pages.cs.aueb.gr/~kintism/papers/tce/

harder, thereby impeding TCE's effectiveness. Hence, we investigate whether the size of the programs or the number of mutants they contain correlate with the effectiveness of the TCE approach.

RQ5 (Size Influence): Does program size or number of mutants affect TCE?

We answer this question by investigating correlations between the number and proportions of both equivalent and duplicated mutants found by TCE with the program and mutant set size.

Finally, since we have results for both C and Java, we investigate the similarities and differences between the two sets of programs. Thus we ask:

RQ6 (Differences between programming languages): What are the similarities and differences between C and Java with respect to TCE?

To answer this question we compare the results of C and Java and try to provide insights on the differences between C and Java as viewed by mutation testing.

3.3 Subject Programs

We used two categories of subject programs for both C and Java. The first category is composed of 6 large to medium open source programs. In this set, we chose 'real-world' programs that vary in size and application domain. The second category of programs was taken from the studies of Yao *et al.* [16] and Kintis and Malevris [49]. We chose these sets because they are accompanied by manually-identified equivalent mutants. The availability of known equivalent mutants allows us to answer RQ3, because it provides a 'ground truth' on the undecidable equivalence question for a set of subjects. The rest of RQs are answered using the larger programs.

Regarding the large programs, compiling all their mutants constitutes a time consuming task. This is due the increase of the mutants according to the size of the programs. It is evident by our reported results, presented in Section 4.2, where it took more than 50 hours to compile only the mutants involved in the *Vim-Eval* component (under *-O3*). TCE may be scalable in itself, but applying it to all possible mutants of a large program is clearly infeasible.

Though there are techniques to reduce the number of mutants, i.e., by sampling, we prefer not to use them in case we unintentionally bias our sample of mutants. We prefer to sample, safer, over the code to be mutated in a systematic way so that we do not pre-exclude any mutants from our investigation. Therefore, in C we rank their source files according to their lines of code. Then, we select the two largest components (source code files). On these two components we apply mutation to all the functions they contain. In Java we followed a similar process by ranking all the project packages according to their size and selected the three largest classes that could be handled without a problem by MUJAVA among the four largest packages.

Tables 2 and 3 respectively present the information about the first category of subject programs for C and Java. Regarding Table 2 (large C subjects), the *Gzip* and *Make* are GNU utility programs. The first program performs file compression and the second one builds automatically executable files from several source code files. The two largest components of *Gzip* are the 'trees' and 'gzip'. The former implements the source representation using variable-length binary code trees and the later implements the main command line interface for the *Gzip* program. The two largest components of the *Make* program are 'main' and 'job'. The later implements utilities for managing individual jobs during the source building processes and the former implements the command line interface. The *GSL* (GNU Scientific Library) is a C/C++ numerical library, which provides a wide range of common mathematical functions. Its two largest components are 'gen' and 'blas'. The 'gen' implements utilities that compute eigenvalues for generalised vectors and matrices. The 'blas' implements BLAS operations for vectors and dense matrices.

The program *MSMTP* is an SMTP client for sending and receiving emails. The components studied are the 'smtp' and the 'msmtp'. The 'smtp' implements the coreutilities for exchanging information with SMTP servers and the 'msmtp' component implements the command line interface.

The program *Git* is a source code management system and the components selected are the 'refs' and 'diff'. The 'refs' implements the 'reference' data structure that associates history edits with SHA-1 values and the 'diff' component implements utilities for checking differences between git objects, for example commits and working trees.

Finally, the program *Vim* is a configurable text editor. The selected components, 'spell' and 'eval', implement utilities for checking and built-in expression evaluation, respectively.

The first two columns of Table 3 (large Java subjects) refer to the first category of programs and their size in terms of source code lines. The domains of the chosen subjects range from mathematics libraries (*Commons-Math*) to build systems (*Ant*). The application domains of the remaining subjects appertain to enhancements of Java's core class (*Commons-Lang*), bytecode manipulation (*BCEL*), date and time manipulation (*Joda-Time*) and database applications (*H2*). Finally, the size of the studied Java programs ranges between 16,753 and 104,479 source code lines. The next two columns of the table present the names of the utilised

TABLE 2: Details of C subjects: 'LoC' shows the lines of code of the project; 'Comp' and 'Comp-Size' show the components considered and their size; 'Func' and 'Muts' show the number of functions and mutants of the components.

Program	LoC	Comp	Comp-Size	Func	Muts
Gzip-1.6	7,323	trees	1,075	14	3,859
		gzip	1,744	26	4,402
MSMTP-1.4.32	13,068	smtp	1,914	23	3,479
		msmtp	4,096	26	9,967
Make 4.0	32,122	main	3,439	11	2,268
		job	3,618	10	2,106
Git-2.1	106,012	refs	3,726	121	6,644
		diff	5,024	125	12,855
GSL-1.16	228,863	gen	2,116	20	7,260
		blas	2,190	106	3,889
Vim-7.2	362,769	spell	16,181	136	33,188
		eval	22,827	374	39,244
Total	**750,157**	**-**	**67,950**	**992**	**129,161**

TABLE 3: Java Test Subjects' details: 'LoC' shows the source code lines of the projects; 'Package' and 'Class-Size' present the packages of the considered classes and their size; the 'Methods' and 'Mutants' columns show the number of methods and the corresponding number of generated mutants.

Program	LoC	Package	Class-Size	Methods	Mutants
Commons-Math-1.2	16,753	org.apache.commons.math.ode	951	34	5,868
		org.apache.commons.math.analysis	429	16	2,861
		org.apache.commons.math.linear	1,294	119	4,962
		org.apache.commons.math.distribution	244	32	546
Commons-Lang-2.4	18,168	org.apache.commons.lang	4,008	350	10,371
		org.apache.commons.lang.builder	967	130	1,661
		org.apache.commons.lang.text	1,915	237	5,983
		org.apache.commons.lang.math	1,247	104	3,999
BCEL-5.2	23,726	org.apache.bcel.generic	1,658	145	2,514
		org.apache.bcel.classfile	897	112	1,065
		org.apache.bcel.verifier.structurals	2,599	351	1,711
		org.apache.bcel.util	974	39	1,666
Joda-Time-2.4	28,255	org.joda.time	1,858	302	2,840
		org.joda.time.format	1,091	90	2,247
		org.joda.time.chrono	487	59	1,723
		org.joda.time.tz	353	36	310
H2-1.0.79	72,359	org.h2.jdbc	4,324	476	3,248
		org.h2.command	4,707	163	4,666
		org.h2.expression	1,130	55	1,774
		org.h2.tools	2,177	277	3,058
Ant-1.8.4	104,479	org.apache.tools.ant.taskdefs	2,035	177	1,194
		org.apache.tools.ant	2,349	163	1,635
		org.apache.tools.ant.types	1,354	91	583
		org.apache.tools.util	1,388	110	1,698
Total	**263,740**	**-**	**40,436**	**3,668**	**68,183**

packages and the size of the considered classes, respectively. Finally, the last two columns of the table refer to the number of methods that belong to the examined classes and the number of the generated mutants.

The second category of subjects contains 8 C and 6 Java programs. The C programs have lines of code ranging from 10 to 42 lines, 7 programs with 137 to 564 lines and 3 real-world programs with 9,564 to 35,545 lines. Additional details for these programs can be found in the work of Yao *et al.* [16]. Details regarding the Java programs are given in Table 4. The first two columns of the table present the examined programs and the considered methods. *Bisect* is a simple program that calculates square roots, `Commons-Lang` and `Joda-Time` are enhancements to java core library and time manipulation libraries, *Pamvotis* is a wireless LAN simulator, *Triangle* is the classic triangle classification program and *XStream* is an XML object serialisation framework. The last two columns of the table present the number of the generated and manually-identified equivalent mutants. It is noted that for the purposes of the present study we extended the original set of programs by manually analysing approximately 400 additional mutants. Thus, in total the considered set is composed of 1,542 manually analysed mutants, out of which 196 are equivalent.

TABLE 4: Manually-analysed Java test subjects' details: 'Program' and 'Method' columns present the examined programs and the considered methods; 'Mutants' shows the number of the generated mutants and 'Equivalent' the number of the manually-identified equivalent ones.

Program	Method	Mutants	Equivalent
Bisect	`sqrt`	135	17
Commons-Lang	`capitalize`	69	14
	`wrap`	198	19
Joda-Time	`add`	257	37
Pamvotis	`addNode`	318	33
	`removeNode`	55	7
Triangle	`classify`	354	40
XStream	`decodeName`	156	29
TOTAL	**-**	**1,542**	**196**

3.4 Mutant Operators

Based on previous research on mutant operator selection, we identify and use two sets of operators (one for C and one for Java). The C set of operators was based on the studies of Offutt *et al.* [51] and Andrews *et al.* [4], [64] and it is composed of 10 operators. A detailed description of the operators is reported in Table 5.

We detail exactly how these operators were applied since this is an important piece of information that differs from one tool to another. The ABS and UOI operators were only applied to numerical variables. The CRCR was applied to integer and floating numeric constants. No mutant operator was applied to the variables of the lefthand side of assignment statements; we only apply them to the right hand sides. This is an implementation choice that avoids the generation of duplicated mutants (as any variable on the

TABLE 5: Mutant operators of MILU.

Name	Description
ABS: *Absolute Value Insertion*	$\{(e,\texttt{abs}(e)),(e,\texttt{-abs}(e))\}$
AOR: *Arithmetic Operator Replacement*	$\{(op_1,op_2) \mid op_1,op_2 \in \{+,-,*,/,\%\} \wedge op_1 \neq op_2\}$
LCR: *Logical Connector Replacement*	$\{(op_1,op_2) \mid op_1,op_2 \in \{\&\&, \vert\vert\} \wedge op_1 \neq op_2\}$
ROR: *Relational Operator Replacement*	$\{(op_1,op_2) \mid op_1,op_2 \in \{>,>=,<,<=,==,!=\} \wedge op_1 \neq op_2\}$
UOI: *Unary Operator Insertion*	$\{(v,--v),(v,v--),(v,++v),(v,v++)\}$
CRCR: *Integer constant replacement*	$\{(c_i,x) \mid x \in \{1,-1,\ 0,\ c_i+1,\ c_i-1,-c_i\}\}$
OAAA: *Arithmetic assignment mutation*	$\{(op_1,op_2) \mid op_1,op_2 \in \{+=,-=,*=,/=,\%=\} \wedge op_1 \neq op_2\}$
OBBN: *Bitwise operator mutation*	$\{(op_1,op_2) \mid op_1,op_2 \in \{\&, \vert\} \wedge op_1 \neq op_2\}$
OCNG: *Logical context negation*	$\{(e,!(e)) \mid e \in \{\texttt{if(e)},\texttt{while(e)}\}\}$
SSDL: *Statement Deletion*	$\{(s,\texttt{remove}(s))\}$

TABLE 6: Mutant operators of MUJAVA.

Name	Description
AORB: *Arithmetic Operator Replacement Binary*	$\{(op_1,op_2) \mid op_1,op_2 \in \{+,-,*,/,\%\} \wedge op_1 \neq op_2\}$
AORS: *Arithmetic Operator Replacement Short-Cut*	$\{(op_1,op_2) \mid op_1,op_2 \in \{++,--\} \wedge op_1 \neq op_2\}$
AOIU: *Arithmetic Operator Insertion Unary*	$\{(v,-v)\}$
AOIS: *Arithmetic Operator Insertion Short-cut*	$\{(v,--v),(v,v--),(v,++v),(v,v++)\}$
AODU: *Arithmetic Operator Deletion Unary*	$\{(+v,v),(-v,v)\}$
AODS: *Arithmetic Operator Deletion Short-cut*	$\{(--v,v),(v--,v),(++v,v),(v++,v)\}$
ROR: *Relational Operator Replacement*	$\{((\mathrm{a}\ op\ \mathrm{b}),\ \texttt{false}),((\mathrm{a}\ op\ \mathrm{b}),\ \texttt{true}),(op_1,\ op_2) \mid op_1,op_2 \in \{>,>=,<,<=,==,!=\} \wedge op_1 \neq op_2\}$
COR: *Conditional Operator Replacement*	$\{(op_1,op_2) \mid op_1,op_2 \in \{\&\&, \vert\vert, ^\wedge\} \wedge op_1 \neq op_2\}$
COD: *Conditional Operator Deletion*	$\{(!cond,cond)\}$
COI: *Conditional Operator Insertion*	$\{(cond,!cond)\}$
SOR: *Shift Operator Replacement*	$\{(op_1,op_2) \mid op_1,op_2 \in \{>>,>>>,<<\} \wedge op_1 \neq op_2\}$
LOR: *Logical Operator Replacement*	$\{(op_1,op_2) \mid op_1,op_2 \in \{\&, \vert, ^\wedge\} \wedge op_1 \neq op_2\}$
LOI: *Logical Operator Insertion*	$\{(v,\sim v)\}$
LOD: *Logical Operator Deletion*	$\{(\sim v,v)\}$
ASRS: *Short-Cut Assignment Operator Replacement*	$\{(op_1,op_2) \mid op_1,op_2 \in \{+=,-=,*=,/=,\%=,\&=, \vert=,^\wedge=,>>=,>>>=,<<=\} \wedge op_1 \neq op_2\}$

lefthand side of assignment statements will be used (and mutated) later in the program). All operators are applied recursively to all sub expressions.

With respect to the Java programming language, we used all the method-level operators of MUJAVA (version 3) [28]. This means that we excluded all the object oriented related mutation operators. Previous research [65] has shown that object oriented mutation operators produce a small number of mutants and a rather low number of equivalent ones and thus, there is no need to investigate this case. MUJAVA supports a wide range of mutant operators built based on the experience and studies of Offutt and colleagues, i.e., [28] and [51].

Table 6 describes the employed mutant operators: the first column of the table presents their names and the second one the mutation they impose. In total, 15 mutant operators were utilised which fall into 6 general categories: arithmetic operators, relational operators, conditional operators, shift operators, logical operators, and assignment operators.

We use these operators due to their extensive use in literature [2]. To generate the C mutants, we use MILU [66], and for the Java mutants, MUJAVA (version 3).

Further details and the implementation of the tools and their operators can be found at the webpages of MILU[8] and MUJAVA[9].

3.5 Experimental Environment

Two series of experiments were conducted. The first one was for programs written in C and the second one for programs written in Java. All the experiments of the C programs were undertaken on the Microsoft Azure Cloud platform using a A9 Compute Intensive Instance in the Ubuntu 14.04 operating system with `gcc 4.8` compiler. To compile the mutants we used four configuration options. We compile with no optimisation settings, denoted as *None*, and with the three popular ones, as realised by the `gcc` compiler, denoted as *-O*, *-O2* and *-O3* . We use the Linux time utility to measure the CPU execution time of all the involved processes. To check whether two binaries are equivalent we use the 'diff' utility with the flag '--binary'. In short, we use a `gcc -flag`' combined with a 'diff'.

All the experiments regarding the Java language were performed on a physical machine running Fedora 22, equipped with an i7 processor (3.40 GHz, 4 cores) and 16GB of memory. TCE relies on compiler optimisation to detect mutant equivalences. While in programming languages such as C or C++ many optimisation options have been embedded within the language compilers, e.g. `gcc`, this does not hold true for the standard Java compiler, i.e. `javac`. Despite the fact that `javac` does not possess advanced optimisation capabilities at the compilation time, it is able to detect some mutant equivalences.

In order to successfully apply TCE to Java, compiler optimisations are required. To this end, we used SOOT [67], a popular framework for analysing and transforming Java applications. SOOT implements various analysis

8. https://github.com/yuejia/Milu/tree/develop/src/mutators
9. https://cs.gmu.edu/~offutt/mujava/

and transformation procedures.We utilised the *-O* option of the tool which performs intra-procedural optimisations. Such optimisations include the 'elimination of common sub-expressions' and 'copy and constant propagation', among others. As in the case of the C language, we used the `diff` command line tool, for the purposes of comparing the optimised classes.

4 TCE VIA GCC

This section reports the results pertaining to the C programming language. Sections 4.1 and 4.2 respectively present results regarding the TCE effectiveness and efficiency. Sections 4.3 and 4.4 detail our results regarding the ground truth and the mutant operators. Finally, Section 4.5 investigates the impact of program size to TCE.

4.1 `gcc`: TCE Effectiveness

To assess the effectiveness of the TCE approach, answering RQ1, we measure the number of the detected equivalent and duplicated mutants. We also measure the proportions of these mutants per program, computed as the percentage of the detected to introduced. When mutants are mutually equivalent to each other, i.e., they are duplicated, one of them should be kept, while, the other(s) should be discarded. In our results we only report the number of mutants that should be discarded.

Table 7 reports our results per program and per considered optimisation option. Overall, these results indicate that TCE can detect in total 9,551 equivalent mutants, accounting for 7.4% of all mutants. TCE also detected 27,163 duplicated mutants, which account for 21% of all mutants. Overall, TCE can thus identify and remove approximately 28% of all mutants as being useless.

Figure 1 depicts the proportions of both equivalent and duplicated mutants detected per program. The horizontal axis of the graph is ordered by the size of the components while the vertical axis records the proportions of mutants detected. From these results, it is evident that all the subjects have a reasonably high proportion of equivalent and duplicated mutants. The proportions of equivalent mutants detected varies from program to program. In the worst case it is 2%, while in the best, 17%. We observe a small variation in the proportions of the identified equivalent and duplicated mutants. The only exceptions are the *Gsl-Blas* and *Gsl-Gen* components. In the former case, TCE detects many equivalent mutants and very few duplicated ones, while, in the later case, it detects very few equivalent mutants and a similar to the other programs ratio of duplicated mutants. This divergence is mainly attributed to the internal structure and code characteristic of the component.

Finally, Table 7 reveals that, depending on the options used, the detected equivalences differ. For instance, the *-O3* option found on average 84% and 100% of the equivalent and duplicated mutants that are detected by applying all the options. Interestingly, with respect to equivalent mutants, among the different optimisation options, i.e., *-O*, *-O2* and *-O3*, there is no clear winner and their behaviour varies between programs. However, the overall differences between the options are relatively small. With respect to duplicated mutants, the results are clear and they show that the best options are the *-O2* and *-O3*.

4.2 `gcc`: TCE Efficiency

To assess the efficiency of the TCE approach and answer RQ2, we report the CPU execution time. Table 8 summarises the execution time of TCE in total, average and per employed component, using the four studied compiler settings. The columns 'Comp.', 'Eq.D.' and 'D.D.' record the execution time with respect to the compilation process, the equivalent mutant detection and duplicated mutant detection, per considered compilation option, respectively.

These results reveal that the execution time of the equivalence detection process is reasonably small compared to the compilation one. For instance, TCE requires on average 22 seconds, for all cases, to detect equivalent mutants, while, the average compilation cost is 5,942 seconds in the best case.

A similar case arises when considering the costs for detecting duplicated mutants. While this is approximately an order of magnitude higher than the cost of detecting equivalent mutants, it is still reasonable; 225 seconds, and no more than 1/30 of the cheapest compilation cost. It is noted that our approach checks for equivalences only for the combinations of mutants that are located on the same function. Therefore, the reported time is analogous to the number of combinations between the mutants located at each function of the project and not between the whole combinations of all project mutants.

Our results show that the compilation time of the *-O3* option is almost 5 times higher than the *None* option. However, this is counterbalanced by the improved effectiveness of the option. In this case, the total time spend for compiling, detecting equivalent and duplicated mutants is 374,162, 260 and 2,744 seconds, respectively. Therefore, TCE analyzed 129,161 mutants in 377,166 seconds. This time accounts for less than 3 seconds per mutant suggesting that its application is reasonable.

4.3 `gcc`: Equivalent Mutants

To determine the ratio of detected to all existing equivalent mutants, we applied TCE to the equivalent mutants identified by Yao *et al.* [16], using the accompanying website data [10]. This site is regularly updated, so data may differ slightly from those previously reported [16]. Additional details about these data can be found on the website.

Table 9 reports the number and the proportions of equivalent mutants detected by TCE when using the different settings. The results are surprisingly good. They reveal that out

10. www0.cs.ucl.ac.uk/staff/Y.Jia/projects/equivalent_mutants/

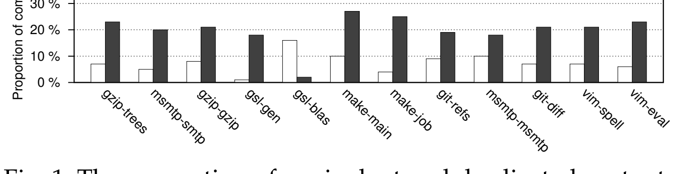


Fig. 1: The proportion of equivalent and duplicated mutants detected by TCE per studied C program.

TABLE 7: Equivalent and duplicated mutants detected by TCE via `gcc`. 'None', '-O', '-O2' and '-O3' report the fraction of all identified equivalent mutants that were detected per optimisation flag. '#Mutants' reports the distinct number of detected mutants by all the options together and '% of all Mutants' reports the percentage of detected to the number of mutants.

Program	None		-O		-O2		-O3		#Mutants		% of all Mutants	
	Eq.	Dup.	Eq.	Dup.	Eq.	Dup.	Eq.	Dup.	Eq.	Dup.	Eq.	Dup.
Gzip–Gzip	0.58	0.85	0.92	0.97	0.94	0.99	0.96	1.00	353	942	8%	21%
Gzip–Trees	0.42	0.60	0.73	0.90	0.97	0.99	0.96	1.00	302	910	8%	24%
Vim–Spell	0.33	0.72	0.76	0.92	0.93	1.00	0.87	1.00	2493	7113	8%	21%
Vim–Eval	0.49	0.83	0.88	0.92	0.61	0.99	0.63	1.00	2570	9028	7%	23%
Make–Main	0.28	0.97	0.56	1.00	0.95	0.97	0.95	0.97	236	625	10%	27%
Make–Job	0.47	0.87	0.85	0.95	0.90	0.98	1.00	1.00	101	529	5%	25%
Git–Diff	0.43	0.85	0.85	0.97	0.92	0.99	0.97	1.00	921	2755	7%	21%
Git–Refs	0.42	0.83	0.84	0.96	0.94	0.99	0.97	1.00	602	1282	9%	19%
Msmtp–Msmtp	0.66	0.72	0.95	0.86	0.73	0.97	0.76	1.00	1017	1835	10%	18%
Msmtp–Smtp	0.33	0.79	0.97	0.96	0.96	0.99	0.97	1.00	178	696	5%	20%
Gsl–Blas	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	651	102	17%	3%
Gsl–Gen	0.66	0.93	0.96	0.99	0.97	1.00	0.95	0.99	127	1346	2%	19%
Total	**0.49**	**0.80**	**0.86**	**0.94**	**0.83**	**0.99**	**0.84**	**1.00**	**9,551**	**27,163**	**7%**	**21%**

TABLE 8: Execution time, measured in sec.: compilation 'Comp.', equivalent mutant detection 'Eq.D.' and duplicated mutant detection 'D.D.'.

Program	Optimisation settings for `gcc`											
	None			-O			-O2			-O3		
	Comp.	Eq.D.	D.D.	Comp.	Eq.D.	D.D.	Comp.	Eq.D.	D.D.	Comp.	Eq.D.	D.D.
	sec			sec			sec			sec		
Gzip–Trees	269	8	112	532	8	231	747	8	165	1,217	8	183
Msmtp–Smtp	405	7	180	743	7	163	1,085	7	201	1,145	7	197
Gzip–Gzip	496	8	230	941	9	212	1,444	9	236	1,578	9	230
Gsl–Gen	1,352	14	193	2,663	14	215	3,945	15	196	3,988	15	212
Gsl–Blas	814	7	58	1,318	7	53	1,864	7	61	1,914	7	59
Make–Main	322	4	138	654	5	155	994	5	148	1,099	5	139
Make–Job	243	4	112	488	4	93	747	4	89	997	4	133
Git–Refs	2,087	13	243	4,038	14	201	5,878	13	232	10,432	14	226
Msmtp–Msmtp	1,929	21	251	3,801	21	274	6,019	21	218	6,751	21	266
Git–Diff	5,662	27	516	11,015	26	470	17,650	25	446	22,318	26	399
Vim–Spell	20,832	65	348	51,475	65	304	85,313	65	327	142,218	67	335
Vim–Eval	36,890	79	287	81,981	81	266	132,888	77	408	180,505	77	365
Total	**71,301**	**257**	**2,668**	**159,649**	**261**	**2,637**	**258,574**	**256**	**2,727**	**374,162**	**260**	**2,744**
Average	**5,942**	**22**	**222**	**13,304**	**22**	**220**	**21,548**	**21**	**227**	**31,180**	**22**	**229**

of all the existing equivalent mutants, TCE can detect from 9% to 100% (with 30% on the average case) of them. With respect to the total number of mutants (killable and equivalent ones), the TCE equivalent ones are approximately 7%. These results are achieved within a few seconds with the potential to save considerable manual and computational resources. Together with the previously presented results, we conclude that TCE is effective and practically applicable on large real-world programs.

Regarding the types of the equivalent detected mutants, i.e. second part of RQ3, we recall that equivalent mutants are equivalent because: a) they reside in unreachable code, b) it is impossible to affect the program state that pertains immediately after mutant execution or c) there is no possible way to propagate the infection they introduce to the program output. Interestingly, the equivalent mutants detected by TCE reside within all of these categories. In particular, TCE detected 6%, 25% and 45% of the equivalent mutants caused by a), b), and c), respectively.

4.4 `gcc`: Mutant Operators

To determine the influence of the mutant operators on the effectiveness of TCE, answering RQ4, we measure the number of detected equivalent and duplicated mutants per operator. We also measure the ratios of detected to introduced mutants by the studied operators. It is noted that the choice of which mutants should be discarded when computing the duplicated mutants, can unfairly influence the reported numbers with respect to the mutant operators that they belong to. To avoid this, in this section we report the number and proportions of all the mutants that are duplicated and not the discarded ones.

Table 10 reports the number and proportions of the equivalent and duplicated mutants found by TCE per pro-

TABLE 9: TCE applied to Yao *et al.* [16] benchmark set: Number 'No.' and proportion '%' of detected equivalent mutants.

Program	None		-O		-O2		-O3	
	No.	%	No.	%	No.	%	No.	%
Min	0	0%	7	78%	9	100%	9	100%
Bubble	0	0%	2	22%	4	44%	2	22%
Profit	0	0%	24	52%	24	52%	24	52%
Mid	0	0%	14	74%	14	74%	14	74%
Prime	0	0%	2	22%	6	67%	6	67%
Triangle	0	0%	16	40%	16	40%	16	40%
Insert	0	0%	11	58%	7	37%	7	37%
Day	3	21%	6	43%	7	50%	7	50%
Calendar	0	0%	12	39%	14	45%	14	45%
Carsimulator	0	0%	33	75%	33	75%	33	75%
Tcas	7	8%	7	8%	8	9%	8	9%
Defroster	16	11%	20	14%	20	14%	20	14%
Schedule	0	0%	14	29%	15	31%	15	31%
Hashmap	0	0%	18	27%	18	27%	18	27%
Replace	29	13%	29	13%	29	13%	29	13%
Space	17	20%	22	25%	26	30%	27	31%
Flex	8	20%	9	23%	12	30%	12	30%
Make	21	35%	39	65%	39	65%	39	65%
Total	**101**	**10%**	**285**	**29%**	**301**	**30%**	**300**	**30%**

gram and operator. These results suggest that on different programs a similar proportion of equivalent and duplicated mutants can be detected by TCE. The only exceptions are the *Gsl-Blas* and *Gsl-Gen* components.

Figure 2 depicts the proportions of equivalent and duplicated mutants detected per operator. The horizontal axis follows the presentation order of the operators from Table 10, while, the vertical axis records the proportions of detected mutants. These results reveal that the ABS and UOI operators introduce at least 15% equivalent mutants of all that they introduce. They also show that TCE detects more than 5% of equivalent mutants produced by the ABS, ROR, UOI and CRCR operators. Regarding the duplicated mutants, TCE detects large proportions, above 10%, on all of them but, the ABS, LCR and OAAA. Interestingly, the LCR operator seems to produce very few equivalent or duplicated mutants.

In conclusion, our results show that all but the LCR and OAAA operators produce a relatively high ratio of useless mutants, i.e., equivalent and duplicated. In practice this involves a huge overhead that, fortunately, can be saved by TCE.

4.5 gcc: Program Size and Mutant Equivalences

To answer RQ5, we use the Spearman rank correlation coefficient ρ. This is a non-parametric statistical test that measures whether two variables' ranks are related, i.e., it assesses the monotonic relationship between the two variables. The Spearman correlation gives values in the range of [-1, +1] with 0 indicating no relationship and +1 indicating a perfect one (-1, also implies a perfect inverse relationship). In addition to the the correlation coefficient ρ we report the obtained p-values that represent the chance that we would observe the ρ value reported, were there, in fact, to be no correlation.

We found a correlation between the number of mutants and the number of equivalent mutants detected ($\rho = 0.818$, $p - value = 0.002$). This suggests that more mutants lead to more equivalent ones. Similarly, a strong correlation between the number of mutants and the detected duplicated ones ($\rho = 0.930$, $p - value < 2.2e - 16$) was also found. The correlation between the number of mutants and the proportion of TCE equivalent and duplicated mutants was found to be ($\rho = -0.091, p - value = 0.783$) and ($\rho = 0.280$, $p - value = 0.379$) respectively. These results suggest that we have no evidence supporting the argument that mutants' number can have a strong influence on the proportions of the equivalences detected by TCE.

We also study the relation between the program size with the number of detected equivalences. We found a medium to small correlation in case of equivalent mutants ($\rho = 0.692$, $p - value = 0.016$). A slightly lower correlation was found between the size of program and the number of duplicated mutants ($\rho = 0.650$, $p - value = 0.026$). With respect to proportions, i.e., correlation between the program size and the proportion of the detected equivalences, we found ($\rho = -0.035$, $p - value = 0.921$) and ($\rho = 0.084$, $p - value = 0.800$) for the cases of equivalent and duplicated mutants, which indicate that we have no data supporting the argument that program size impacts the ratios of the detected equivalences.

Finally, we found a medium correlation between the size of program and the whole number of mutants ($\rho = 0.671$, $p - value = 0.020$), which indicates that larger programs have more mutants than smaller ones. In conclusion, we find no evidence of any correlation between the ratios of equivalent and duplicated mutants in any of the size indicators. This means there is no evidence that the proportion goes up or down as the size of the program or the number of mutants changes. However there is evidence that the number goes up with the size, as one would expect. Taken together based on the studied mutant set, these can be regarded as evidence suggesting that the number of TCE equivalent and duplicated mutants is a fairly consistent proportion, unaffected by the size of the program. These results may be explained by the fact that the compiler optimisations we use only apply "locally", i.e., on the occurrences of code patterns, and not on the semantic of the entire system.

5 TCE via javac and Soot

This section details our results for Java. Sections 5.1 and 5.2 respectively present results regarding TCE effectiveness and Efficiency. Sections 5.3 and 5.4 detail our results regarding

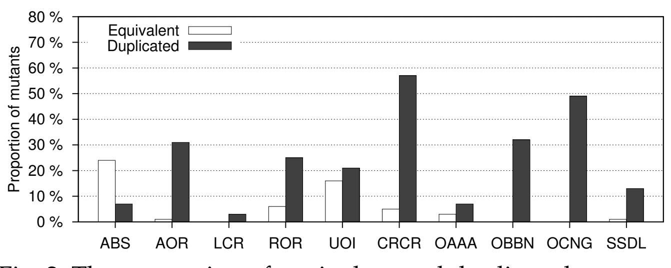


Fig. 2: The proportion of equivalent and duplicated mutants detected by TCE per mutant operator in case of C.

TABLE 10: Number 'No.' and proportion '%' of equivalent and duplicated mutants detected by TCE per operator.

Equivalent Mutants																				
Program	ABS		AOR		LCR		ROR		UOI		CRCR		OAAA		OBBN		OCNG		SSDL	
	No.	%	No.	%	No.	%	No.	%	No.	%	No.	%	No.	%	No.	%	No.	%	No.	%
Gzip–Trees	111	27%	3	0%	0	0%	44	9%	74	9%	39	2%	0	0%	0	0%	1	1%	30	11%
Msmtp–Smtp	43	27%	3	1%	0	0%	7	1%	102	32%	19	1%	0	0%	0	0%	0	0%	4	0%
Gzip–Gzip	42	27%	1	0%	10	16%	37	4%	70	22%	141	6%	0	0%	1	2%	6	2%	45	7%
Gsl–Gen	24	2%	6	0%	0	0%	22	4%	14	21%	59	1%	0	0%	0	–	0	0%	2	0%
Gsl–Blas	0	0%	0	–	0	0%	0	0%	0	–	650	50%	0	–	0	–	0	0%	1	0%
Make–Main	26	59%	26	9%	0	0%	38	10%	14	15%	104	9%	22	26%	0	0%	3	3%	3	1%
Make–Job	22	22%	0	0%	0	0%	16	4%	32	16%	27	2%	0	0%	0	0%	0	0%	4	1%
Git–Refs	131	27%	0	0%	0	0%	19	2%	184	19%	260	8%	0	0%	0	0%	0	0%	8	0%
Msmtp–Msmtp	131	20%	8	3%	0	0%	7	0%	216	17%	645	14%	0	0%	0	0%	0	0%	10	0%
Git–Diff	189	22%	4	0%	0	0%	63	4%	328	19%	327	5%	0	0%	0	0%	0	0%	10	0%
Vim–Spell	832	26%	47	2%	0	0%	476	8%	760	12%	353	3%	9	3%	0	0%	0	0%	16	0%
Vim–Eval	671	32%	8	0%	0	0%	697	7%	836	20%	331	1%	0	0%	0	0%	0	0%	27	0%
Total	**2,222**	**24%**	**106**	**1%**	**10**	**0%**	**1,426**	**6%**	**2,630**	**16%**	**2,955**	**5%**	**31**	**3%**	**1**	**0%**	**10**	**0%**	**160**	**1%**

Duplicated Mutants																				
Program	ABS		AOR		— LCR —		ROR —		UOI —		CRCR		OAAA		— OBBN		OCNG		SSDL	
	No.	%	No.	%	No.	%	No.	%	No.	%	No.	%	No.	%	No.	%	No.	%	No.	%
Gzip–Trees	24	5%	139	42%	2	20%	193	42%	239	29%	738	50%	8	10%	4	80%	44	80%	65	25%
Msmtp–Smtp	6	3%	66	35%	3	6%	108	18%	46	14%	831	50%	0	0%	3	25%	71	63%	67	16%
Gzip–Gzip	8	5%	25	19%	1	1%	178	22%	74	24%	1193	56%	0	0%	17	41%	97	47%	88	14%
Gsl–Gen	28	3%	418	21%	0	0%	88	18%	8	12%	1691	55%	3	3%	0	–	39	33%	53	8%
Gsl–Blas	0	0%	0	–	0	0%	88	5%	0	–	0	0%	0	–	0	–	88	67%	28	4%
Make–Main	0	0%	145	54%	0	0%	90	24%	9	10%	701	64%	0	0%	0	0%	34	41%	24	10%
Make–Job	3	3%	41	66%	2	5%	84	21%	33	16%	636	68%	0	0%	5	31%	41	47%	40	14%
Git–Refs	25	5%	76	46%	4	3%	170	21%	184	19%	1654	56%	0	0%	27	50%	70	24%	61	6%
Msmtp–Msmtp	17	2%	95	43%	7	4%	188	13%	257	20%	2026	45%	13	13%	9	33%	149	35%	300	22%
Git–Diff	35	4%	95	20%	8	4%	357	23%	313	18%	3494	59%	11	18%	47	35%	142	23%	165	11%
Vim–Spell	353	11%	730	32%	16	4%	1888	31%	1306	21%	6809	59%	22	8%	23	27%	533	60%	504	18%
Vim–Eval	124	5%	329	38%	16	2%	2503	28%	1036	24%	10793	62%	1	0%	13	17%	882	61%	430	11%
Total	**623**	**7%**	**2,159**	**31%**	**59**	**3%**	**5,935**	**25%**	**3,505**	**21%**	**30,566**	**57%**	**58**	**7%**	**148**	**32%**	**2,190**	**49%**	**1,825**	**13%**

the ground truth and the mutant operators. Finally, section 5.5 investigates the impact of program size on TCE.

5.1 `javac` and SOOT: TCE Effectiveness

In an analogous manner to the results of Section 4.1, we present our findings that are pertinent to RQ1, i.e. the effectiveness of TCE. These results are illustrated in Table 11 and Figure 3.

Table 11 presents the equivalent and duplicated mutants detected by `javac` and SOOT per test subject. The '#Mutants' column, which is divided into the 'Eq.' and 'Dup' sub-columns, presents the number of the detected equivalent and duplicated mutants (per tool). The '% of all Mutants' column records their corresponding proportion to the generated mutants. From the depicted results, it is clear that SOOT outperforms `javac` in both equivalent and duplicated mutant detection, managing to detect 3,904 equivalent mutants and 3,687 duplicated ones. Thus, code optimisations implemented in SOOT appear to be superior to the ones of `javac`. Furthermore, it should be mentioned that the mutants detected by SOOT form a superset of the ones detected by `javac`. Therefore, we conclude that SOOT constitutes an appropriate tool for TCE.

It is noted that most Java-to-bytecode compilers mainly perform runtime optimizations than static ones. Thus, class files are optimised by the Java virtual machine as they are interpreted and not at the compilation time. This explains why the Java stock compiler is infective.

Figure 3 illustrates the proportion of equivalent and duplicated mutants per test subject. The horizontal axis presents the corresponding proportions and the vertical presents the test subjects in ascending order, according to their size. By examining the figure, it becomes evident that TCE manages to detect a considerable number of equivalent and duplicated mutants, ranging between 1% and 18% for equivalent ones and 2% and 17% for duplicated ones.

To summarise, in the case of Java, TCE managed to detect 6% of all mutants as equivalent and 5% of them as duplicated ones.

5.2 `javac` and SOOT: TCE Efficiency

In this section, we detail the empirical findings pertaining to TCE's efficiency for the case of Java. To this end, Table 12 presents the CPU execution time that the equivalent and duplicated detection required per test subject and optimisation tool.

Table 12 is divided into three columns: 'Program' refers to the names of test subjects; the '`javac`' column reports the compilation time ('Comp.' sub-column), the equivalent mutant detection time ('Eq.D.' sub-column) and the duplicated mutant detection time ('D.D.' sub-column) of TCE via `javac`; and, 'SOOT' presents the corresponding results in the case of TCE via SOOT. It should be noted that in

TABLE 11: Equivalent and duplicated mutants detected by TCE via `javac` and SOOT.

Program	javac				SOOT			
	#Mutants		% of all Mutants		#Mutants		% of all Mutants	
	Eq.	Dup.	Eq.	Dup.	Eq.	Dup.	Eq.	Dup.
org.apache.commons.math.ode	2	63	0%	1%	150	110	3%	2%
org.apache.commons.math.analysis	0	25	0%	1%	192	91	7%	3%
org.apache.commons.math.linear	0	90	0%	2%	82	140	2%	3%
org.apache.commons.math.distribution	0	8	0%	1%	64	14	12%	3%
org.apache.commons.lang	0	645	0%	6%	498	697	5%	7%
org.apache.commons.lang.builder	20	108	1%	7%	84	110	5%	7%
org.apache.commons.lang.text	0	265	0%	4%	364	339	6%	6%
org.apache.commons.lang.math	8	171	0%	4%	260	199	7%	5%
org.apache.bcel.generic	1	92	0%	4%	157	108	6%	4%
org.apache.bcel.classfile	0	27	0%	3%	54	27	5%	3%
org.apache.bcel.verifier.structurals	1	256	0%	15%	29	257	2%	15%
org.apache.bcel.util	0	30	0%	2%	104	35	6%	2%
org.joda.time	5	118	0%	4%	511	129	18%	5%
org.joda.time.format	6	97	0%	4%	156	151	7%	7%
org.joda.time.chrono	0	39	0%	2%	184	75	11%	4%
org.joda.time.tz	2	21	1%	7%	26	25	8%	8%
org.h2.jdbc	11	138	0%	4%	433	150	13%	5%
org.h2.command	30	210	1%	5%	214	240	5%	5%
org.h2.expression	20	124	1%	7%	78	129	4%	7%
org.h2.tools	1	115	0%	4%	134	161	4%	5%
org.apache.tools.ant.taskdefs	14	123	1%	10%	24	127	2%	11%
org.apache.tools.ant	1	157	0%	10%	44	162	3%	10%
org.apache.tools.ant.types	2	100	0%	17%	4	101	1%	17%
org.apache.tools.util	0	101	0%	6%	58	110	3%	6%
Total	**124**	**3,123**	**0%**	**5%**	**3,904**	**3,687**	**6%**	**5%**

this column the reported compilation time also includes the execution time of the tool. Finally, the last two rows of the table present the total and average time of the examined analyses.

Regarding equivalent mutant detection, TCE via `javac` required in total 140 seconds to detect 124 equivalent mutants, while, TCE via SOOT required 208 seconds for the identification of 3,904 equivalent mutants, with an average of 6 and 9 seconds per examined package, respectively. Given that the corresponding compilation time is 2,703 seconds for `javac` and 78,318 seconds (compilation and optimisation) for SOOT, with an average of 113 and 3,263, it can be argued that TCE forms a practical approach for detecting equivalent mutants. Considering the duplicated mutants, the application of `javac` detected 3,123 mutants in 13,636 seconds and SOOT detected 3,687 mutants in 19,399 seconds. It is noted that both equivalent and duplicated mutants that are detected by `javac` form a subset of those detected by SOOT. Overall, the presented data suggest that TCE manages to automatically and safely discard a significant number of useless mutants in only a small fraction of the time, in less than 2 seconds per examined mutant.

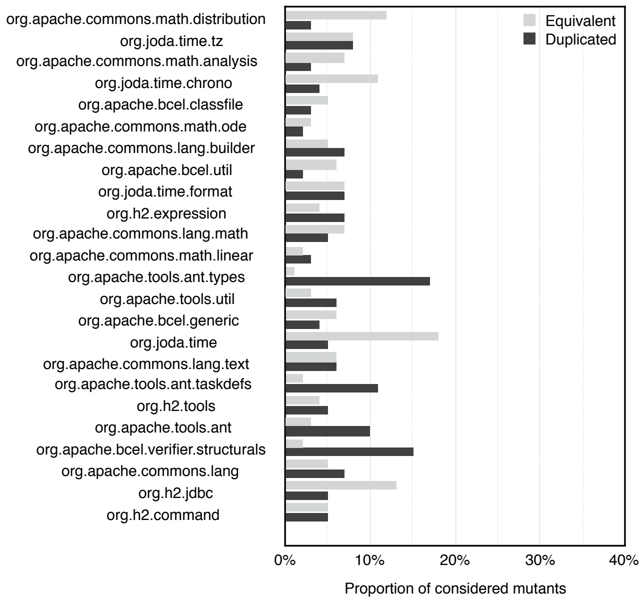


Fig. 3: The proportion of equivalent and duplicated mutants detected by TCE per studied Java program.

The question that is raised here is whether the time required by TCE is acceptable. While this depends on many uncontrolled parameters, we would like to underline that detecting equivalent mutants is a tedious and manual task. Previous research estimated the time of the manual identification of a single equivalent mutant to be approximately 15 minutes [40]. Assuming this is a fair approximation, identifying the TCE equivalent mutants pure manually would require 124×15 minutes or 111,600 seconds for the case of `javac` and $3,904 \times 15$ minutes or 3,513,600 seconds for the case of SOOT. Thus, it can be easily concluded that the execution cost of TCE is small when compared to

TABLE 12: Execution time, measured in sec., of equivalent and duplicated mutant detection per considered tool and test subject.

Program	`javac`			SOOT		
	Comp.	Eq.D.	D.D.	Comp. & Opt.	Eq.D.	D.D.
	sec			sec		
org.apache.commons.math.ode	82	11	4,376	5,583	16	7,065
org.apache.commons.math.analysis	27	6	1,277	2,194	8	2,022
org.apache.commons.math.linear	90	9	633	5,060	14	1,029
org.apache.commons.math.distribution	9	2	22	381	2	34
org.apache.commons.lang	995	18	903	13,930	26	1,319
org.apache.commons.lang.builder	45	4	112	1,480	5	184
org.apache.commons.lang.text	152	12	665	6,847	17	960
org.apache.commons.lang.math	57	7	582	3,658	12	923
org.apache.bcel.generic	62	5	185	2,530	7	242
org.apache.bcel.classfile	26	2	54	991	3	77
org.apache.bcel.verifier.structurals	138	4	67	2,673	6	110
org.apache.bcel.util	45	4	484	1,529	5	702
org.joda.time	65	6	102	2,923	8	126
org.joda.time.format	37	5	291	1,997	7	397
org.joda.time.chrono	21	4	266	1,348	5	379
org.joda.time.tz	9	1	12	301	2	17
org.h2.jdbc	165	8	365	4,870	21	424
org.h2.command	349	9	2,011	8,656	14	1,721
org.h2.expression	39	4	179	1,645	5	247
org.h2.tools	98	6	715	3,887	8	934
org.apache.tools.ant.taskdefs	46	3	51	1,367	4	78
org.apache.tools.ant	64	4	91	1,966	5	132
org.apache.tools.ant.types	26	2	21	615	3	33
org.apache.tools.util	56	4	172	1,887	5	244
Total	**2,703**	**140**	**13,636**	**78,318**	**208**	**19,399**
Average	**113**	**6**	**568**	**3,263**	**9**	**808**

the estimated manual effort. In fact, the total cost of TCE (optimisation phase + detection phase) constitutes only 3% of the estimated manual effort.

5.3 `javac` and SOOT: Equivalent Mutants

This section, which answers RQ3, provides insights into the actual proportion of equivalent mutants that can be automatically detected by TCE. We perform this evaluation based on manually-identified sets of such mutants to gives us a ground truth. Table 13 describes the corresponding findings per utilised tool. As can be seen, `javac` failed to detect equivalent mutants on our ground truth benchmark. By contrast, SOOT detected 105 out of 196 equivalent mutants, indicating that it can automatically weed out more than 50% of the studied equivalent mutants. These results provide strong evidence regarding the TCE's effectiveness. Finally, it should be stated that these automatically-detected equivalent mutants correspond to 7% of all the studied ones, which is in line with the results of the large-scale experiment we report in Section 5.1.

A manual analysis of the types of equivalent mutants that are TCE equivalent reveals that all but one of the detected mutants belong to the third category, i.e., the corresponding mutant can be reached and can infect the program state locally but subsequently fail to propagate the corrupted state to the observable output. The one mutant not falling into this category is a mutant that can be reached but not infected.

TABLE 13: TCE applied to Java benchmark set: Number 'No.' and proportion '%' of detected equivalent mutants.

Method	`javac`		SOOT	
	No.	%	No.	%
sqrt	0	0%	11	65%
capitalize	0	0%	2	14%
wrap	0	0%	12	63%
add	0	0%	22	59%
addNode	0	0%	31	94%
removeNode	0	0%	6	86%
classify	0	0%	21	52%
decodeName	0	0%	0	0%
Total	**0**	**0%**	**105**	**54%**

5.4 `javac` and SOOT: Mutant Operators

In order to answer RQ4 for Java, this section reports the contribution of each mutant operator to the detected mutants. More precisely, Table 14 presents the number of the detected equivalent mutants per operator, along with their proportion to all the generated mutants by that specific operator, and Table 15 presents the respective results for the case of the duplicated mutants. For brevity, we only record the cases that the number of detected mutants was higher than 0 in the studied programs.

By examining Table 14, it becomes clear that TCE (via SOOT) managed to detect equivalent mutants that belong to 7 out of the 15 utilised mutant operators, indicating that it can be effective across a wide range of operators. With respect to the duplicated mutants discovered, the findings of Table 15 show that these mutants belong to 8 operators, corroborating the previous statement.

Figure 4 visualises the proportions of detected mutants across the corresponding mutant operators. It can be seen that TCE manages to identify at least 4% of the equivalent mutants produced by LOR, AODS and AOIS and at least 24% of the duplicated ones generated by ROR and COI. Again, for brevity reasons we depict only those operators with higher than 0% detection rates.

It is noted that the TCE equivalences are a special form of redundancy as they require mutual subsumption between mutants (mutant a subsumes mutant b and mutant b subsumes mutant a). This is different from the redundant mutants studied by Kaminski *et al.* [59] and Just *et al.* [60] which consider non-mutual subsumptions (mutant a subsumes b and mutant b does not subsumes a). In view of this, it is normal that the COR operator produces redundant mutants that are not captured by TCE (results reported in Tables 14 and 15). Still, a stronger version of the COR operator may provide more chances for TCE equivalences.

5.5 `javac` and SOOT: Program Size and Mutant Equivalences

In order to answer RQ5, i.e., whether or not the number of generated mutants or the program size affects TCE, we examined the correlation between the program size and the number and proportions of the detected equivalent and duplicated mutants. All ρ values, and p-values, were computed using the Spearman rank correlation test.

Regarding the correlation between the number of mutants and the number of identified equivalent ones, a ρ of 0.786, $p - value = 8.536e - 06$, was obtained, indicating a strong correlation. The correlation between the number of mutants and the proportion of the equivalent ones was found to be ρ of -0.008, $p - value = 0.972$.

In the case of duplicated mutants, i.e., correlation between the number of mutants and the number of duplicated ones, $\rho = 0.657$, $p - value = 0.001$ and $\rho = -0.271$, $p - value = 0.199$ with respect to number and proportions of duplicated mutants detected. Based on these data, it can be concluded that the number of equivalent and duplicated mutants detected by TCE tends to increase as the number of the generated mutants increases. However, this does not appear to be the case when considering the detected proportions.

With respect to the correlation of program size with the detected equivalent and duplicated mutants, the obtained results suggest that there is a very weak correlation in the case of the equivalent mutants ($\rho = 0.230$, $p - value = 0.277$) whereas, in the case of the duplicated ones, there is a strong one, i.e., $\rho = 0.797$, $p - value = 3.081e - 06$. The correlations between program size and proportion of detected equivalent and duplicated mutants was found to be $\rho = -0.337, p{-}value = 0.107$ and $\rho = 0.423, p{-}value = 0.041$. It is noted that this last case, i.e., program size and duplicated mutants, is the only one where our data show a correlation.

In conclusion, our data show that an increase in the program size is expected to increase the number of equivalent and duplicated mutants identified by TCE. However, the proportion of equivalent mutants detected is expected to be unaffected by the program size, while the proportion of duplicated ones is affected. Finally, we found a low but nontrivial correlation between the program size and the number of generated mutants, i.e., $\rho = 0.417$, with p-value = 0.044.

6 Discussion

This section summarise our results and concludes the stated RQs. It also discusses the practical implications and constraints of applying mutation with the use of TCE.

6.1 Results Summary

6.1.1 TCE Effectiveness

Our results suggest that TCE can reduce the total number of mutants by 11% for Java and 28% for C. In the case of C, TCE equivalent mutants range from 2% to 17% depending on the studied program and account for 7.4% of all mutants on average. In the case of Java, TCE, using Soot, revealed 5.7% equivalent mutants, on average, that range from 1% to 18%. TCE duplicated mutants range from 3% to 27% and account for 21.0% on average when considering C, while for Java, they range from 2% to 17% and they are 5.4% on average.

6.1.2 TCE Efficiency

The time to detect equivalent and duplicated mutants, using the diff utility, ranges between programs and it is on average 22 and 225 seconds for C and 9 and 808 seconds for Java. This indicates that once the mutants have been compiled/optimised, the equivalence detection comes 'almost for free'. This is an important finding because it suggests that TCE can be applied to remove equivalent and duplicated mutants before the application of other time consuming cost-reduction methods.

Our results show that the total time spent for compiling, detecting equivalent and duplicated mutants is 374,162 and 95,222 seconds for C and Java respectively. Thus, a candidate mutant can be analyzed by TCE in less than 3.0 and 1.5 seconds for C and Java respectively.

6.1.3 Equivalent Mutants

In an attempt to identify the prevalence of TCE equivalent mutants we estimated their ratio, with respect to all equivalent mutants, based on the studied benchmarks. We found that approximately 30% and 54% of the benchmark mutants are trivially equivalent with respect to C and Java. Here it should be noted that there is a large variation on the detected ratios among the studied programs. This is common for both C and Java subjects, indicating that program characteristics have a strong influence on the TCE equivalences.

TABLE 14: Number 'No.' and proportion '%' of equivalent mutants detected by TCE per operator.

Program	AODU		AOIS		AOIU		AORB		AORS		ASRS		LOR	
	No.	%	No.	%	No.	%	No.	%	No.	%	No.	%	No.	%
org.apache.commons.math.ode	0	0%	148	6%	0	0%	2	0%	0	0%	0	0%	0	0%
org.apache.commons.math.analysis	0	0%	192	13%	0	0%	0	0%	0	0%	0	0%	0	0%
org.apache.commons.math.linear	0	0%	82	3%	0	0%	0	0%	0	0%	0	0%	0	0%
org.apache.commons.math.distribution	0	0%	64	28%	0	0%	0	0%	0	0%	0	0%	0	0%
org.apache.commons.lang	2	5%	494	14%	2	0%	0	0%	0	0%	0	0%	0	0%
org.apache.commons.lang.builder	20	44%	64	12%	0	0%	0	0%	0	0%	0	0%	0	0%
org.apache.commons.lang.text	0	0%	364	14%	0	0%	0	0%	0	0%	0	0%	0	0%
org.apache.commons.lang.math	4	17%	252	17%	4	2%	0	0%	0	0%	0	0%	0	0%
org.apache.bcel.generic	0	0%	156	17%	0	0%	0	0%	0	0%	0	0%	1	25%
org.apache.bcel.classfile	0	0%	54	11%	0	0%	0	0%	0	0%	0	0%	0	0%
org.apache.bcel.verifier.structurals	0	0%	28	10%	0	0%	0	0%	0	0%	0	0%	1	25%
org.apache.bcel.util	0	0%	102	16%	0	0%	0	0%	2	6%	0	0%	0	0%
org.joda.time	0	0%	506	51%	5	1%	0	0%	0	0%	0	0%	0	0%
org.joda.time.format	0	0%	150	19%	6	4%	0	0%	0	0%	0	0%	0	0%
org.joda.time.chrono	0	0%	184	25%	0	0%	0	0%	0	0%	0	0%	0	0%
org.joda.time.tz	0	0%	24	24%	2	8%	0	0%	0	0%	0	0%	0	0%
org.h2.jdbc	0	0%	422	33%	11	3%	0	0%	0	0%	0	0%	0	0%
org.h2.command	0	0%	184	14%	29	5%	1	1%	0	0%	0	0%	0	0%
org.h2.expression	0	0%	58	14%	20	9%	0	0%	0	0%	0	0%	0	0%
org.h2.tools	0	0%	130	10%	2	1%	0	0%	0	0%	2	7%	0	0%
org.apache.tools.ant.taskdefs	0	0%	10	5%	14	19%	0	0%	0	0%	0	0%	0	0%
org.apache.tools.ant	1	100%	42	11%	1	1%	0	0%	0	0%	0	0%	0	0%
org.apache.tools.ant.types	0	0%	2	2%	2	7%	0	0%	0	0%	0	0%	0	0%
org.apache.tools.util	0	0%	58	12%	0	0%	0	0%	0	0%	0	0%	0	0%
Total	**27**	**11%**	**3,770**	**15%**	**98**	**2%**	**3**	**0%**	**2**	**0%**	**2**	**0%**	**2**	**4%**

TABLE 15: Number 'No.' and proportion '%' of duplicated mutants detected by TCE per operator.

Program	AODS		AOIS		AOIU		AORB		COD		COI		ROR		SOR	
	No.	%	No.	%	No.	%	No.	%	No.	%	No.	%	No.	%	No.	%
org.apache.commons.math.ode	0	0%	96	4%	0	0%	24	1%	0	0%	47	24%	47	11%	0	0%
org.apache.commons.math.analysis	0	0%	106	7%	0	0%	0	0%	0	0%	25	31%	25	7%	0	0%
org.apache.commons.math.linear	0	0%	100	4%	0	0%	0	0%	0	0%	90	36%	90	15%	0	0%
org.apache.commons.math.distribution	0	0%	12	5%	0	0%	0	0%	0	0%	8	27%	8	8%	0	0%
org.apache.commons.lang	0	0%	102	3%	0	0%	0	0%	0	0%	643	63%	643	25%	0	0%
org.apache.commons.lang.builder	0	0%	4	1%	0	0%	0	0%	0	0%	108	57%	108	27%	0	0%
org.apache.commons.lang.text	0	0%	206	8%	0	0%	0	0%	0	0%	235	68%	235	21%	0	0%
org.apache.commons.lang.math	0	0%	48	3%	2	1%	0	0%	0	0%	171	56%	171	17%	0	0%
org.apache.bcel.generic	0	0%	32	3%	0	0%	0	0%	0	0%	92	63%	92	24%	0	0%
org.apache.bcel.classfile	0	0%	0	0%	0	0%	0	0%	0	0%	27	37%	27	27%	0	0%
org.apache.bcel.verifier.structurals	0	0%	2	1%	0	0%	0	0%	1	1%	255	77%	256	45%	0	0%
org.apache.bcel.util	0	0%	10	2%	0	0%	2	1%	0	0%	29	37%	29	19%	0	0%
org.joda.time	0	0%	18	2%	0	0%	0	0%	0	0%	118	87%	118	25%	0	0%
org.joda.time.format	0	0%	70	9%	0	0%	0	0%	0	0%	97	61%	97	21%	0	0%
org.joda.time.chrono	0	0%	70	9%	0	0%	0	0%	0	0%	39	71%	39	18%	0	0%
org.joda.time.tz	0	0%	8	8%	0	0%	0	0%	0	0%	20	69%	20	36%	2	33%
org.h2.jdbc	0	0%	24	2%	0	0%	0	0%	0	0%	138	63%	138	30%	0	0%
org.h2.command	1	1%	56	4%	0	0%	8	5%	0	0%	206	44%	206	20%	0	0%
org.h2.expression	0	0%	10	2%	0	0%	0	0%	1	5%	123	59%	124	31%	0	0%
org.h2.tools	0	0%	66	5%	6	4%	0	0%	0	0%	115	52%	115	23%	0	0%
org.apache.tools.ant.taskdefs	0	0%	8	4%	0	0%	0	0%	0	0%	123	44%	123	42%	0	0%
org.apache.tools.ant	0	0%	10	3%	0	0%	0	0%	0	0%	157	48%	157	46%	0	0%
org.apache.tools.ant.types	0	0%	2	2%	0	0%	0	0%	0	0%	100	65%	100	54%	0	0%
org.apache.tools.util	0	0%	21	4%	0	0%	0	0%	0	0%	99	47%	99	20%	0	0%
Total	**1**	**1%**	**1,081**	**4%**	**8**	**0%**	**34**	**1%**	**2**	**0%**	**3,065**	**56%**	**3,067**	**24%**	**2**	**4%**

Another important finding regards the causes of mutant equivalences that are detected by TCE. Our results are surprising since they show that the majority of detected mutants are due to failed propagation, i.e, there is no possible way to propagate the mutant infection to the program output. This is true for both C and Java. In Java almost all, 99%, of the detected mutants are of this category, while in C these are 57%. In the case of C, 41% of the detected mutants fall in the second category, i.e, it is impossible to affect the program state that pertains immediately after mutant execution, and 2% to the first one, i.e., mutants reside in unreachable code.

6.1.4 Mutant Operators

To better understand the nature of TCE mutants we identified their prevalence according to the considered mutant operators. Our results suggest that in C the ABS and UOI operators introduce more than 15% of trivial equivalent mutants, ROR and CRCR more than 5% and OAAA just 3% while, LCR, OBBN, AOR, OCNG and SSDL introduce a small fraction, less than 1%. Regarding Java most of the detected equivalent mutants are due to AOIS, 15%, and AODU, 11%. Also, LOR and AOIU introduce notable numbers that respectively account for 4% and 2%. The rest of the operators introduce none or non-significant numbers.

With respect to duplicated mutants, all operators introduce a large number of such mutants in C. Most of them, account for more than 7%. Only LCR introduces a smaller fraction that is 3%. In the case of Java, the situation is a bit different. Only COI and ROR operators have large proportions of TCE duplicated mutants. These are 56% and 24% for COI and ROR. AOIS also produces a large number of duplicated mutants which accounts for 4%. The rest of the operators introduce none or small numbers.

6.1.5 Program Size and Mutant Equivalences

We measured the correlation between the number of mutants and the size of programs. Our results reveal that in both cases there is medium level correlation which is stronger for C, i.e., $\rho = 0.671$ for C and $\rho = 0.417$ for Java. Thus, programs of similar size can vary much in terms of number of mutants. By measuring the average number mutants per statement we get 1.90 and 1.69 for C and Java respectively. Hence, for the programs we studied, we conclude that C programs have approximately 10% more mutants and a stronger correlation, between mutants number and program lines of code, than the Java ones.

With respect to equivalent mutants, our results indicate a strong correlation with the number of mutants, for both C and Java, i.e., $\rho = 0.818$ and $\rho = 0.786$. This is getting weaker when considering program size, i.e., $\rho = 0.692$ for C and $\rho = 0.230$ for Java. However, in all cases we found no evidence indicating that the ratio of the detected equivalent mutants correlates with the number of mutants. Together these two results can be regarded as evidence suggesting that the number of the detected equivalent mutants is a fairly consistent proportion, unaffected by the size indicators of the program under analysis.

With respect to duplicated mutants, our results suggest a strong correlation with the number of mutants, for both C and Java, i.e., $\rho = 0.930$ and $\rho = 0.657$. However, both in C and Java we found no evidence indicating that the ratio of the duplicated mutants correlates with the number of mutants. Program size has medium to strong correlation with the number of TCE duplicated mutants, i.e., $\rho = 0.650$ and $\rho = 0.797$ for C and Java. In case of C we found no evidence indicating that the ratio of the duplicated mutants correlates with program size. In contrast a medium level correlation was found in the case of Java, $\rho = 0.423$.

6.2 Differences between C and Java

Our presentation this far has focused on our results as found by the two versions of TCE, i.e., for C and Java. Here we attempt to compare the results of the C with Java versions, answering RQ6, and highlight commonalities and differences between them.

One first observation is that TCE detects more equivalences in C than in Java. This can be attributed to the compiler optimisations implemented in `gcc` that are way more advanced than that of Java and SOOT. We took a close look at the analysis on the detected causes of equivalence and found that almost all TCE equivalent mutants detected in Java programs are those that cannot propagate, while, only the 57% of the C ones are due to the same reason. This suggests, that there is a 42% difference between the results of C and Java, mainly due to the lack of Java optimisations. The average detected ratios are 7.4% and 5.7%, for C and Java, that reflects the mentioned differences.

Our results demonstrate that equivalent mutants are more prevalent in C than in Java. This is evident from our ground truth analysis which revealed that in C the equivalent mutants account for 23%, while, in Java for 12.7% of all mutants. Additionally, Java has a larger number of trivially equivalent mutants. This is also shown by our ground truth analysis, which revealed that 54% of all Java equivalent mutants are TCE equivalent. The same ratio for C is 30%. In this result, we should consider our first observation, i.e., that 42% of the TCE equivalent mutants cannot be detected by SOOT due to lack of compiler optimizations, that a potentially high number of Java trivially equivalent mutants exists but not found by SOOT. Thus, we can easily conclude that Java programs have considerably less equivalent mutants than the C ones and at the same time Java programs contain a much larger proportion of trivially equivalent mutants.

Regarding duplicated mutants, we found TCE duplicated mutants in C are more prevalent than in Java programs. As our results shown that while in C a large proportion, of 21.0% on average, exists, in Java these mutants are considerably less and account for 5.4% on average. This difference is partly attributed to the lack of optimisations in Java and to language characteristics. Thus, characteristics, like the distinction of logical and arithmetic operators in Java, the typed conventions that are stronger in Java than in C and the use of pointers and arrays make C mutants more vulnerable to duplication.

Another interesting point is that after removing the TCE equivalent mutants, a ratio of 5.8% of equivalent mutants remain in java, while in C the ratio of equivalent mutants that remain is 16.1%. Considering this observation together with the one regarding the number of mutants, that are approximately 10% less in Java than in C, we conclude that,

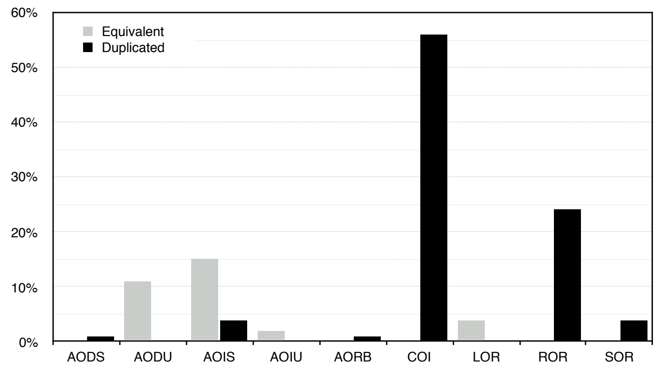


Fig. 4: The proportion of equivalent and duplicated mutants detected by TCE per program studied in case of Java.

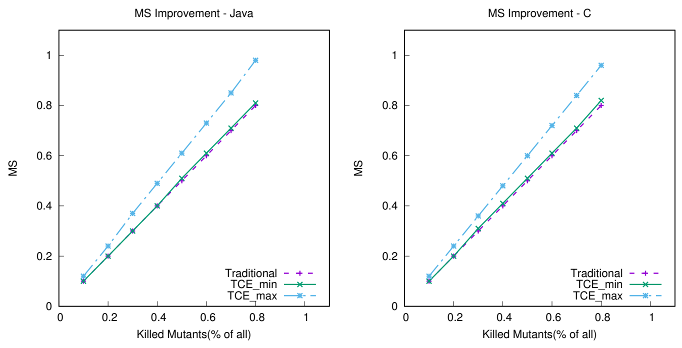


Fig. 5: Mutation score improvements by TCE, when no manual analysis of equivalent mutants has been performed, e.g. in large-scale experiments.

based on the programs we studied, mutation analysis in C is harder than in Java.

The efficiency differences between C and Java in detecting duplicated mutants is believed to be due to the language differences. Our results suggest that 0.028 sec are required per C mutant under analysis while 0.28 per Java one. C binary code tends to be smaller than Java bytecode. While the differences are not practically significant, these could be ameliorated by using some form of checksum, as done by md5 to improve substantially the performance of the diff comparisons.

Considering other parameters, like the tools and operator sets used, could also lead to the differences in C and Java results. While, in C we have 10 operators and in Java 15, this difference is more conventional than actual. It is noted that the CRCR operator corresponds to many Java operators mainly due to the language differences, i.e. in C there are only arithmetic values while in Java logical operations are strictly of boolean types. Only two C operators, the ABS and SDL, are only partially implemented in Java; ABS is partially implemented by AODU and SDL by the various deletion operators like the COD. Three Java operators, SOR, AODS, and AORS, are not implemented in C.

Comparing individual operators, C-ABS produces 24% of TCE equivalent mutants while Java-AODU 11%. Similarly, C-UOI 16% while Java-AOIS 15%. Interestingly, C-ROR, C-CRCR and C-OAAA account for 6%, 5% and 3% respectively while their Java version for 0%. With respect to duplicated mutants, C-ROR produces 25% while the Java-ROR 24%. C-OCNG produces 49% and the Java-COI 56%. C-UOI produces 21% and the Java-AOIS 4%. All other C operators introduce many duplicated mutants not detected by the related Java ones. A manual inspection of the detected C mutants suggests that most of these mutants are due to a failed infection, i.e., mutant execution cannot result in a corrupted program state. As shown by our results, Java optimizations are ineffective for these cases and hence we get a reduced effectiveness.

6.3 Implications for Research Studies

Our results have direct implications for research studies: the application of TCE can improve the accuracy of a study's results when no manual analysis of equivalent mutants have been performed. To better understand these implications, Figure 5 illustrates the range in which TCE can change the resulting mutation scores in the case of Java (left part of the figure) and C (right part of the figure), when assuming that our results are generalisable. Both parts present the mutation scores with no manual analysis (line "traditional") and the improved mutation scores that could be obtained by applying TCE. We report the minimum and maximum number of detected equivalent mutants (lines "TCE_min" and "TCE_max") to better reflect the impact of TCE. Note that the minimum and maximum values are based on the results of our large-scale experiment (see also Sections 4.1 and 5.1). By examining the figures, it can be seen that TCE can improve the accuracy of the obtained mutation scores. More precisely, in the case of Java, this improvement ranges between 0%-18% and, in the case of C, it ranges between 0%-16%.

While these results are only illustrative and have to be treated with a great deal of caution, they provide evidence that research studies will benefit from the application of TCE, by automatically improving the accuracy of the results reported. Consider for instance a study that compares two test generation methods, say methods X and Z which achieve a mutation score (without the analysis of equivalent mutants) of 60% and 67% respectively, and the study concludes that Z is better because it manages to achieve a better mutation score of 67% with an improvement of 7% over the previous method. TCE can be used to improve the accuracy of the study's results: by applying TCE, the mutation score of X will range between 61% and 73% and the one of Z, between 68% and 82%. Thus, the application of TCE will result in more accurate mutation scores and will potentially reveal a greater difference, of 9%, between X and Z, improving the empirical evidence of Z's superiority.

6.4 Practical Implications

Practitioners use test criteria to develop test suites and to assess the level of test thoroughness. Thus, in practice, TCE affects the effort needed (required work) to develop test suites and the ability of the criterion to accurately measure the effectiveness of the test suites. This section investigates these two practical implications of TCE by examining its impact on the *work* required, when generating mutation adequate test suites, and by examining the *improvements* it makes when measuring the mutation score.

TABLE 16: TCE applied to Siemens suite and PROTEUM mutants (benchmark set by Papadakis *et al.* [47]): Number 'No.' and proportion '%' of detected equivalent mutants.

Program	Mutants	Killable	Equivalent		Duplicated	
			No.	%	No.	%
Replace	10,694	8,572	918	8.58%	2431	22.73%
TCAS	2,872	2,357	156	5.43%	482	16.78%
TotInfo	6,411	5,839	177	2.76%	1422	22.18%
Schedule	2,107	1,769	143	6.79%	527	25.01%
Schedule2	2,594	2,068	229	8.83%	535	20.62%
Printtokens	4,266	3,541	295	6.92%	1139	26.70%
Printtokens2	4,574	3,783	377	8.24%	1534	33.54%

To reliably investigate both the required *work* and the *improvements* of TCE we need to know which mutants are equivalent. We also need to have multiple test suites of various levels of test thoroughness, i.e., with low and high mutation scores. The benchmark set of Yao *et al.*, which we use to answer RQ3 (for C programs), is unfortunately short on both of the above two requirements. Thus, we used the benchmark of Papadakis *et al.* [47], which is an extension of the famous Siemens suite [68] and contains manually augmented test suites (mutation adequate) and analysed mutants. This benchmark was constructed using: a) the PROTEUM[11] mutation testing tool to generate mutants, b) manual analysis to characterise these mutants as killable or equivalent and c) manual analysis to augment the test suites (generate tests that kill the identified killable mutants) [47]. In the case of Java, we used the mutation adequate test suites, which we generated when analysing the mutants of the ground truth set (used to answer RQ3).

A summary of the mutants produced by PROTEUM, when applied to the Siemens suite and the results of TCE (using the -O option) are given in Table 16. From these data, it becomes evident that a non-trivial number of mutants has been detected by TCE. The numbers of the TCE equivalent mutants account for 30% - 48% (41% on average) of all the existing equivalent mutants. Interestingly, the results are very similar to those reported in our previous analysis (approximately 6.9% and 24% of all the PROTEUM mutants are TCE equivalent and duplicated) and thus, we are confident that they are representative.

6.4.1 Practical Implications: Required Work

To measure the manual effort involved when performing mutation testing, we adopt the model used by the recent study of Kurtz *et al.* [57]. Thus, we define work as "the number of mutants that are examined by the engineer", or equally, "the sum of the number of tests written to kill all non-equivalent mutants and the number of equivalent mutants identified" [57]. This metric in essence approximates the manual effort that a tester needs to perform when doing mutation testing.

Equation 1 presents the work model. In order to compare the results across different programs, we normalise the recorded work by dividing with the overall required work, per subject. The corresponding formula is presented in Equation 2.

11. We used the version 2.0 of the Proteum/IM tool [27].

$$work = |testCases| + |EquivalentMutants| \qquad (1)$$

$$normalised\_work = \frac{|testCases| + |EquivalentMuts|}{OverallWorkRequired} \qquad (2)$$

Algorithm 1 presents the procedure followed to calculate work, as suggested by Kurtz et al. [57]. First, a mutant is randomly selected from the generated set of mutants of the program under test. Next, if the mutant is equivalent, the work is increased by one and the process is repeated. If the mutant is killable, a test case that kills this mutant is randomly selected, the value of work is increased by one and the other mutants that can be killed by this test case are marked as killed. This process continues until every killable mutant of the considered mutants is selected/killed.

Algorithm 1 Calculating the work metric.

```
    Let muts represent the program's generated mutants
    Let tcs represent the program's mutation test suite
 1: function WORKCALCULATION($muts$, $tcs$)
 2:     $work \leftarrow 0$
 3:     while $\exists\ killable\_mutant \in muts$ do
 4:         $mut \leftarrow$SELECTRANDOMALIVEMUTANT($muts$)
 5:         if ISEQUIVALENT($mut$) then
 6:             $work \leftarrow work + 1$
 7:             CONTINUE
 8:         end if
 9:         $killing\_tc \leftarrow$SELECTRANDKILLINGTC($mut$,$tcs$)
10:         $work \leftarrow work + 1$
11:         UPDATEKILLEDMUTANTS($killing\_tc$, $muts$)
12:     end while
13:     return $work$
14: end function
```

As can be seen from the algorithm, it requires two inputs: a mutant set and a set of mutation adequate test cases. Thus, we calculate the work based on manually analysed test subjects. To avoid any bias from the selection process, we repeated the experiment 100 times.

Figures 6 and 7 illustrate the results obtained for both programming languages. These figures plot the normalised work (x-axis) against the subsuming mutation score [56] (MS^*, y-axis) realised at each step of Algorithm 1 with and without the application of TCE (denoted by the "TCE" and "Traditional" lines respectively) per test subject and programming language. Following the process of Kurtz *et al.* [57], we used the subsuming mutation score as effectiveness measurement. This measurement avoids the inflation effects caused by redundant mutants [56], [57].

By examining Figure 6, it can be seen that TCE manages to substantially reduce the work required to achieve a given test effectiveness level: for instance, in the case of Joda-Time, by applying TCE, the work required to achieve a 70% subsuming mutation score is reduced by 11% compared to the application of mutation without TCE, this reduction increases to approximately 20% when the subsuming mutation score reaches 80% and to 30% when the score reaches 90%; finally, at the 100% effectiveness level, TCE

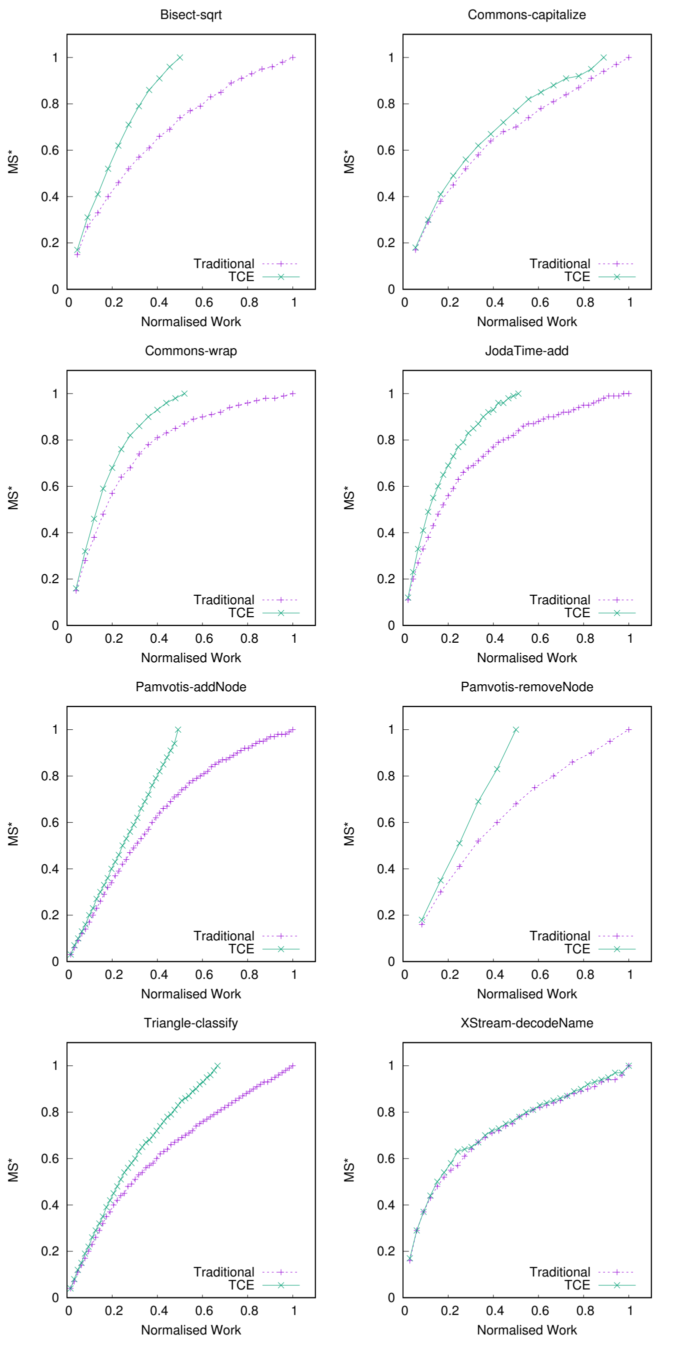


Fig. 6: Work required for different effectiveness levels with and without the application of TCE in the case of Java.

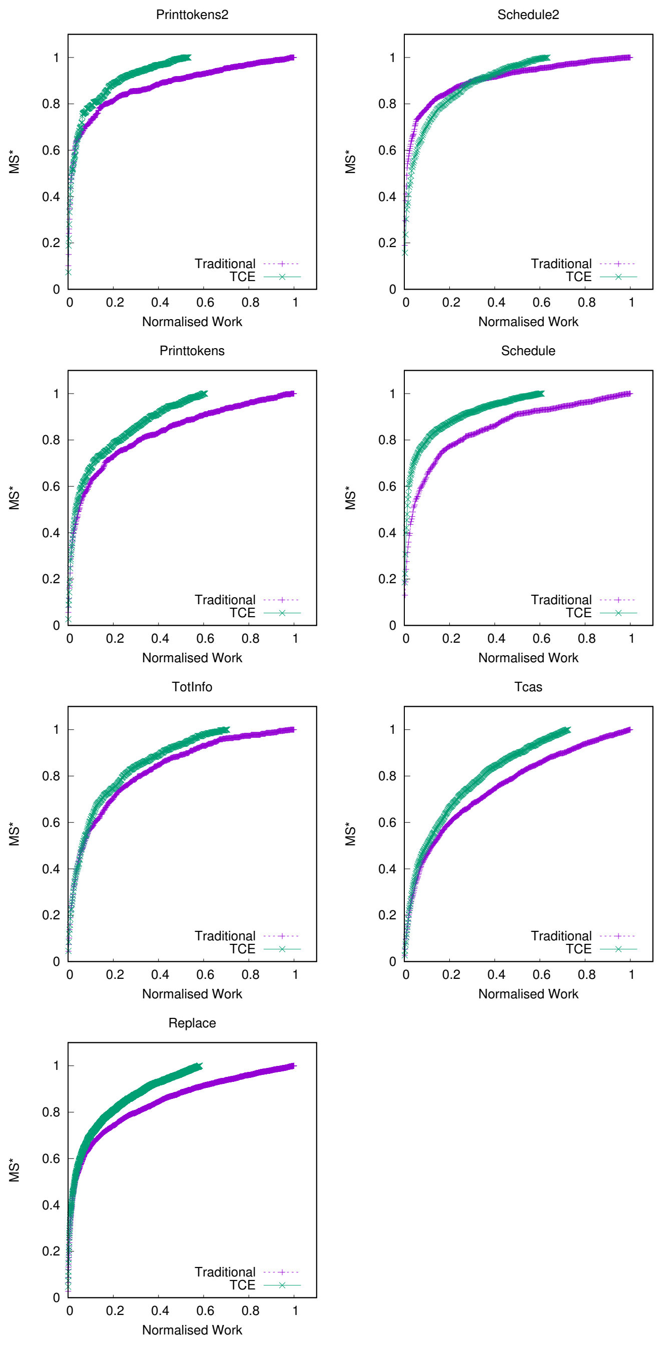


Fig. 7: Work required for different effectiveness levels with and without the application of TCE in the case of C.

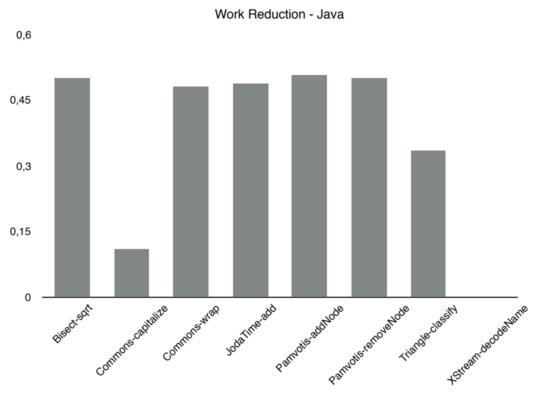
realises a 49% work reduction. This trend, i.e., the increase of the work reduction as the subsuming mutation score increases is present in most Java subjects[12]. This fact can be justified by TCE's equivalent mutant detection which, in turn, gives practitioners a higher chance of selecting killable mutants than equivalent ones, as the application of mutation progresses. Regarding the results for C depicted in Figure 7, it can be seen that analogous conclusions can be drawn.

To better portray TCE's implications for work, Figure 8 presents the overall work reduction when developing mutation adequate test suites per test subject and programming language. It can be seen that the application of TCE realises a work reduction between 0% and 51%, with an average of 37%, in the case of Java and a work reduction that ranges between 28% and 47%, with an average of 37%, in the case of C. These results suggest that the work of an engineer aiming at creating mutation adequate test suites can be substantially reduced by the application of TCE.

12. XStream is a clear outlier but it should be mentioned that it is the only program for which TCE did not detect any equivalent mutant.

6.4.2 Practical Implications: Mutation Score Improvement

This section investigates how the use of TCE improves the accuracy of the mutation score measurement. Consider the following example: an engineer applies mutation to a test



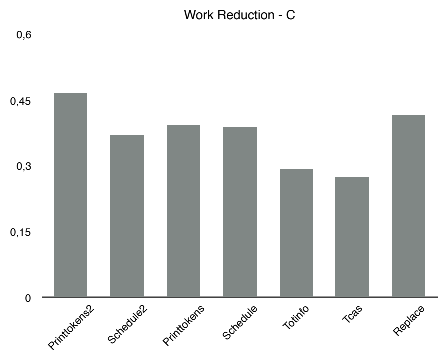


Fig. 8: Overall Work Reduction after the application of TCE per test subject and programming language.

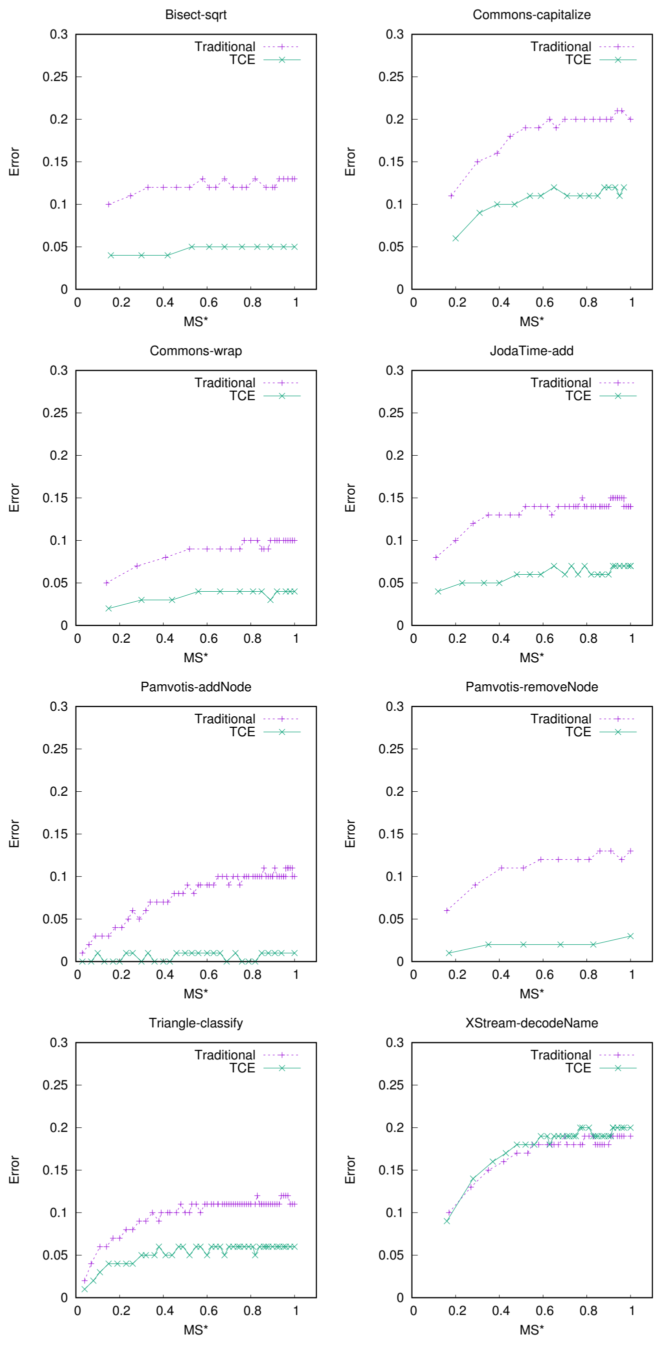


Fig. 9: Mutation score improvement after the application of TCE in the case of Java.

subject based on the available test cases and obtains a value x of mutation score; the question that is raised here is how much does the x score differ from the true mutation score, i.e., the score computed by removing the equivalent mutants?

We calculate the error of the measurement by comparing the true mutation score with the obtained one (with and without applying TCE). Equation 3 details the error of the computation. This metric quantifies the distance of our metric from the true one. Our results are depicted in Figures 9 and 10 for Java and C, respectively. The y-axis of the figures refers to the aforementioned error and the x-axis to the effectiveness levels denoted by the subsuming mutation score.

$$Error = True\_MS - Obtained\_MS \tag{3}$$

By examining Figures 9 and 10, it can be seen that the application of TCE results in a much lower error than calculating the mutation score without its application in most test subjects. For instance, in the case of the `wrap` method of the Commons test subject, at the 75% subsuming mutation score, the error in the mutation score's calculation is 9% without the application of TCE; this error is reduced to 4% when TCE is applied; this difference remain approximately the same until the 100% subsuming mutation score is reached. Overall, in the case of Java, TCE reduces the calculation error of the mutation score by 1%-10%. In the case of C, we find analogous results, with the calculation error reduction ranging between 0% and 4%.

6.5 Application constraints

The proposed technique is solely based on the use of compilers and their optimisation options, thereby avoiding the several limitations of other methods and tools, e.g. applicability and scalability. It does not require any sophisticated source code analysis techniques or any expensive test executions. Thus, it can be directly applied to real-world systems and can be easily incorporated within mutation testing tools.

Interestingly, the detected mutant equivalences are partly dependent on the compiler options used. Although it is rather unlikely that equivalent mutants detected by one compiler option are not equivalent according to another, to

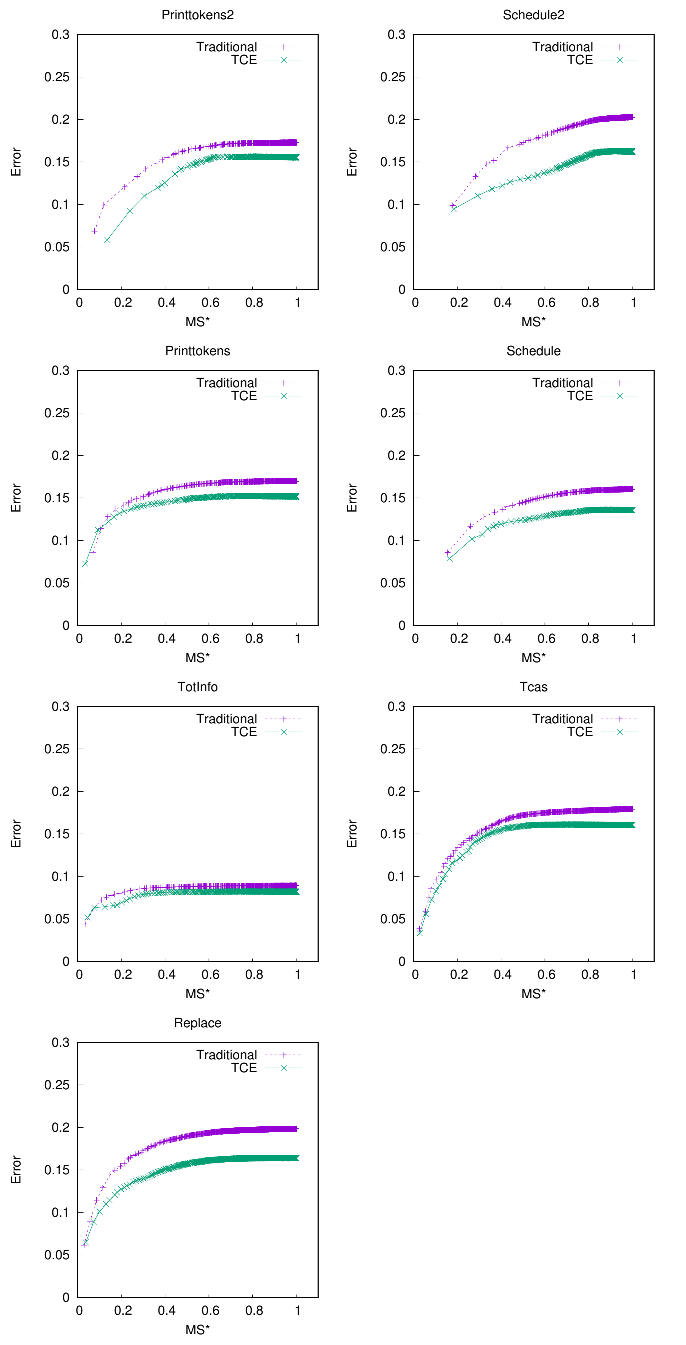


Fig. 10: Mutation score improvement after the application of TCE in the case of C.

be absolutely sure, beyond any doubt that TCE guarantees equivalence, we need to know which compiler settings are going to be used in the deployment environment. No previous research takes into account the particular compiler settings, but since we are using TCE, this cannot be ignored. All previous work implicitly assumes that there is only one compiler option, but actually there are as many options as the actual settings used by the deployed programs. When the deployed-code compiler settings are known, TCE can exploit this information. When they are unknown at mutation test time, we can investigate with a reasonable sample, checking for variance in equivalence behaviour. We investigated this issue by exploring the main `gcc` and SOOT settings covering a wide range of optimisation options and found that all of them can be used to detect mutant equivalences (some are more effective than others of course).

We also explored the trade off between effectiveness and efficiency using different settings. Our results suggest that the *-O* and *-O2* options are reasonably good, because they consume less compilation time than the *-O3* option. However, none of them is superior to the others in detecting equivalent mutants. Here it should be noted that there are many more optimisation options in the modern compilers, there might exist some combinations of them that can detect faster or more mutant equivalences. Thus, our future research is directed towards identifying the options that fit best to TCE. Detailed information about the performed optimizations can be found in the `gcc`[13] and SOOT [14]websites.

7 THREATS TO VALIDITY

As it is usual in software engineering experiments, our subjects might not be representative. It is also possible that they might not hold for complete system analysis (as we only analysed sampled components of the large programs). To ameliorate this issue, we selected 12 real-world programs of varying size and application domain, 6 written in C and 6 written in Java, several orders of magnitude larger than those used in previous equivalent mutant detection studies. We also performed an additional evaluation using different sets of programs, composed of 31 manually-analysed benchmark subjects, taken from the literature. To further cater for this issue, we draw attention to strongly observed effects and present our results as ranges of expected values.

The evaluation of our approach resulted in analogous findings in all studied sets. With reference to the C test subjects, it detected approximately 7.4% of all the mutants as equivalent ones for the large-scale experiment, and 7.2% and 6.9% of the mutants of the manually-analysed test subjects (for the Yao *et al.* [16] and Papadakis *et al.* benchmarks [47]), on average. In the case of Java, it identified 5.7% and 6.8%, accordingly. Regarding the range of the results (range between worst and best cases), a similar picture appears. Thus, we are confident that TCE can eliminate a considerable number of equivalences.

Additionally, our results are in line with those reported in the literature[15] providing confidence that they are realistic. We studied the mutants of the C and Java languages and TCE implemented using `gcc` and SOOT. Therefore, some of our results might be a realisation of independent uncontrolled variables, such as the sample size, sample selection procedure (excluding classes not handled by MUJAVA), programs' internal characteristics, used software platforms and tools' operation. Therefore, it is important to note that all our results form empirical observations that might not hold in the general case. However, our findings fit intuition and rely on the foundations set by previous studies [31]. Furthermore, we control the major factors that we believe

13. https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html
14. https://ssebuild.cased.de/nightly/soot/doc/soot_options.htm
15. Offutt and Pan [33] reported that 9% of all the mutants are equivalent. Delamaro *et al.* [69] found 12% , Kintis *et al.* [46], Schuler and Zeller [40] 7%-8%, Papadakis *et al.* [47] 17%, Yao *et al.* [16] 23% and Madeyski *et al.* [44] 4%-39%.

can influence our results. Additional studies are needed to determine what influences the performance of TCE and its practical use on different languages and compiler optimisation techniques.

Other threats are due to the use of software systems. For instance, the `gcc` compiler or the SOOT optimisation framework may have defects. However, these systems are heavily tested and deployed. Thus, it is unlikely that the remaining defects would influence our results to a great extent. Implementation defects of MILU, MUJAVA and PROTEUM may also have an influence. To reduce this threat we carefully checked their results. However, we consider this as a minor threat since all the used tools have been used by several authors in recent studies, e.g., [6], [57], [70], [71], [72], independently of us. Furthermore, we utilised three equivalent mutant benchmark sets which were entirely built by hand. These results served as a 'sanity check' to reduce the threat to validity.

Our results might be affected by our choice of mutant operators. As shown by other studies [73], [74] it is also possible that the realisation of the mutant operators by the employed tools may particularly affect the comparison between C and Java. To mitigate this threat, we detailed exactly how the operators supported by the used tools are realised (Tables 5 and 6) and analysed the common C and Java operators. Based on these lines we draw some conclusions. Overall, we used a wide range of 75 mutant operators (realised by PROTEUM) and all the popular operators (included in most existing mutation testing tools and those empirically found to correlate with fault detection). In all cases we found large numbers of equivalences, which have a major impact on the application of mutation testing.

The use of the equivalent mutants' benchmarks may also pose another threat. This is due to the performed manual analysis: some killable mutants may have been mistakenly identified as equivalent. However, these studies were performed independently of the present one and hence, it is not likely that this kind of mistakes coincidentally match the results of TCE. Additionally, it is equally possible that such mistakes have also led to the underestimation of TCE's effectiveness.

Finally, all our subjects, tools and data are available in the accompanied website of the present paper[16]. This helps reducing all the above-mentioned threats [75] since independent researchers can check, replicate and analyse our findings.

8 Conclusion and Future Work

We have presented the results of an extensive empirical analysis of the ability of Trivial Compiler Equivalence (TCE) to detect both equivalent and duplicated mutants in the C and Java programming languages.

We have conducted an empirical study of TCE on 25 C and 6 Java benchmark systems, for which the programs under study are sufficiently small for their equivalent mutants to be determined manually. These systems provided us with the ground truth against which we can empirically assess the equivalent mutant detection power of TCE. We augmented this study with a much larger study for which no ground truth is possible. In total, we have experimented with over 1 million lines of code, consisting of the 31 smaller benchmark systems, together with 6 larger Java systems (with a total of 263,740 LoC) and 6 larger C systems (with a total of 750,157 LoC).

Overall, we find that for both C and Java, TCE is a useful, fast and widely-applicable technique that can detect between 17%-100% (30% on average) of C language equivalent mutants, and 0%-94% (54% on average) of Java equivalent mutants (for the ground truth set). Furthermore, over all mutants studied in all large real-world programs, the detection of trivially equivalent and trivially duplicated mutants was found to reduce the total number of mutants by 5%-23% for Java and 20%-37% for C, which accounts for 11% and 28% on average. These achievements imply that a practitioner who applies mutation testing and is using TCE will spend 0%-51% and 28%-47% less manual effort in the case of Java and for C than without using it. TCE also improves the accuracy of the mutation score measurement by 1%-10% and 0%-4% for Java and C. Thus, future research should integrate compiler optimisations within mutation testing tools in order to avoid any generation of such trivial mutants and future research studies should consider applying TCE to reap the benefit of the technique.

Our results revealed interesting findings that suggest topics for future work on mutation-based analysis of the semantic differences between programming languages. For example, it is intriguing that a larger proportion of Java's equivalent mutants were found to be detectable using TCE than for C. Furthermore, if the proportion of equivalent mutants from the ground truth study is similar to that for mutants overall, then it would appear that the Java language suffers significantly less from the equivalent mutant problem than the C language does.

One might conjecture that this is related to the relatively small size of Java methods when compared to the size of C functions. Alternative conjectures might revolve around the differing semantic features of these two languages (and the consequent mutation operators that are applicable). Of course, since we have insufficient data to make scientifically reliable statements on these conjectures, we have refrained from making any claims in the present paper and leave them as just that; conjectures. Nevertheless, our results suggest that future work might use TCE as one approach to tackle such conjectures, potentially leading to a better understanding of the difference between programming language semantics, based on mutation analysis.

Acknowledgments

Marinos Kintis is partly supported by the Research Centre of Athens University of Economics and Business (RC/AUEB). Mike Papadakis is supported by the National Research Fund, Luxembourg, INTER/MOBILITY/14/7562175 and by Microsoft Azure Grant 2015. Mark Harman is partly supported by the UK EPSRC projects EP/I033688 (GISMO) and EP/G060525, a 'Platform Grant' for the Centre for Research on Evolution Search and Testing (CREST) at UCL. Yue Jia is supported by the EPSRC project EP/J017515 (DAASE) and by Microsoft Azure Grant 2014.

16. http://pages.cs.aueb.gr/~kintism/papers/tce/

References

[1] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "The Design of a Prototype Mutation System for Program Testing," in *Proceedings of the AFIPS National Computer Conference*, vol. 74. Anaheim, New Jersey: ACM, 5-8 June 1978, pp. 623–627.

[2] Y. Jia and M. Harman, "An Analysis and Survey of the Development of Mutation Testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, 2011.

[3] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *FSE*, 2014, pp. 654–665.

[4] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is Mutation an Appropriate Tool for Testing Experiments?" in *ICSE*, 2005, pp. 402 – 411.

[5] M. Daran and P. Thévenod-Fosse, "Software error analysis: A real case study involving real faults and mutations," in *ISSTA*, 1996, pp. 158–171.

[6] R. Baker and I. Habli, "An empirical evaluation of mutation testing for improving the test quality of safety-critical software," *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 787–805, 2013.

[7] B. H. Smith and L. Williams, "On guiding the augmentation of an automated test suite via mutation analysis," *Emp. Soft. Eng.*, vol. 14, no. 3, pp. 341–369, 2009.

[8] A. J. Offutt, J. Pan, K. Tewary, and T. Zhang, "An Experimental Evaluation of Data Flow and Mutation Testing," *Software Pract. Exper.*, vol. 26, no. 2, pp. 165–176, 1996.

[9] P. G. Frankl, S. N. Weiss, and C. Hu, "All-uses vs Mutation Testing: an Experimental Comparison of Effectiveness," *Journal of Systems and Software*, vol. 38, no. 3, pp. 235–253, September 1997.

[10] R. A. DeMillo and A. J. Offutt, "Constraint-Based Automatic Test Data Generation," *IEEE Transactions on Software Engineering*, vol. 17, no. 9, pp. 900–910, September 1991.

[11] M. Harman, Y. Jia, and W. B. Langdon, "Strong higher order mutation-based test data generation," in 8^{th} *European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE '11)*. New York, NY, USA: ACM, September 5th - 9th 2011, pp. 212–222. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025144

[12] R. P. Mateo, P. U. Macario, F. Alemán, and J. Luis, "Validating second-order mutation at system level," *IEEE Trans. Softw. Eng.*, vol. 39, no. 4, pp. 570 – 587, April 2013.

[13] Y. Jia and M. Harman, "Constructing Subtle Faults Using Higher Order Mutation Testing," in *SCAM*, 2008, pp. 249–258.

[14] K. Androutsopoulos, D. Clark, H. Dan, M. Harman, and R. Hierons, "An analysis of the relationship between conditional entropy and failed error propagation in software testing," in 36^{th} *International Conference on Software Engineering (ICSE 2014)*, Hyderabad, India, June 2014, pp. 573–583.

[15] T. A. Budd and D. Angluin, "Two Notions of Correctness and Their Relation to Testing," *Acta Informatica*, vol. 18, no. 1, pp. 31–45, 1982.

[16] X. Yao, M. Harman, and Y. Jia, "A Study of Equivalent and Stubborn Mutation Operators Using Human Analysis of Equivalence," in *ICSE*, 2014, pp. 919–930.

[17] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei, "Is operator-based mutant selection superior to random mutant selection?" in *ICSE*, 2010, pp. 435–444.

[18] W. E. Wong and A. P. Mathur, "Reducing the Cost of Mutation Testing: An Empirical Study," *J. Syst. Software*, vol. 31, no. 3, pp. 185–196, December 1995.

[19] M. Papadakis and N. Malevris, "An Empirical Evaluation of the First and Second Order Mutation Testing Strategies," in *ICST Workshops*, 2010, pp. 90–99.

[20] M. Polo, M. Piattini, and I. Garcia-Rodriguez, "Decreasing the Cost of Mutation Testing with Second-Order Mutants," *Softw. Test. Verif. Rel.*, vol. 19, no. 2, pp. 111 – 131, June 2008.

[21] M. H. Bill Langdon and Y. Jia, "Efficient multi objective higher order mutation testing with genetic programming," *Journal of Systems and Software*, vol. 83, no. 12, pp. 2416–2430, July 2010.

[22] M. Harman, Y. Jia, P. R. Mateo, and M. Polo, "Angels and monsters: an empirical investigation of potential test effectiveness and efficiency improvement from strongly subsuming higher order mutation," in *ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. ACM, 2014, pp. 397–408.

[23] M. Papadakis and N. Malevris, "Automatic mutation test case generation via dynamic symbolic execution," in *ISSRE*, 2010, pp. 121–130.

[24] L. Zhang, D. Marinov, and S. Khurshid, "Faster mutation testing inspired by test prioritization and reduction," in *ISSTA*, 2013, pp. 235–245.

[25] P. R. Mateo and M. P. Usaola, "Reducing mutation costs through uncovered mutants," *Softw. Test., Verif. Reliab.*, vol. 25, no. 5-7, pp. 464–489, 2015. [Online]. Available: http://dx.doi.org/10.1002/stvr.1534

[26] M. Papadakis, Y. Jia, M. Harman, and Y. L. Traon, "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique," in *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, 2015, pp. 936–946. [Online]. Available: http://dx.doi.org/10.1109/ICSE.2015.103

[27] M. E. Delamaro, J. C. Maldonado, and A. M. R. Vincenzi, *Proteum/IM 2.0: An Integrated Mutation Testing Environment*. Boston, MA: Springer US, 2001, pp. 91–101. [Online]. Available: http://dx.doi.org/10.1007/978-1-4757-5939-6_17

[28] Y.-S. Ma, A. J. Offutt, and Y.-R. Kwon, "MuJava: An Automated Class Mutation System," *Softw. Test. Verif. & Rel.*, vol. 15, no. 2, pp. 97–133, June 2005.

[29] D. Baldwin and F. G. Sayward, "Heuristics for Determining Equivalence of Program Mutations," Yale University, New Haven, Connecticut, Research Report 276, 1979.

[30] A. T. Acree, "On Mutation," PhD Thesis, Georgia Institute of Technology, Atlanta, Georgia, 1980.

[31] A. J. Offutt and W. M. Craft, "Using Compiler Optimization Techniques to Detect Equivalent Mutants," *Softw. Test. Verif. Rel.*, vol. 4, no. 3, pp. 131–154, 1994.

[32] A. J. Offutt and J. Pan, "Detecting Equivalent Mutants and the Feasible Path Problem," in *COMPASS*, 1996, pp. 224–236.

[33] ——, "Automatically Detecting Equivalent Mutants and Infeasible Paths," *Softw. Test. Verif. Rel.*, vol. 7, no. 3, pp. 165–192, September 1997.

[34] J. Voas and G. McGraw, *Software Fault Injection: Inoculating Programs Against Errors*. John Wiley & Sons, 1997.

[35] R. M. Hierons, M. Harman, and S. Danicic, "Using Program Slicing to Assist in the Detection of Equivalent Mutants," *Softw. Test. Verif. Rel.*, vol. 9, no. 4, pp. 233–262, 1999.

[36] M. Harman, R. Hierons, and S. Danicic, "The relationship between program dependence and mutation analysis," in *Mutation Testing for the New Century*. Kluwer Academic Publishers, 2001, pp. 5–13.

[37] K. Adamopoulos, M. Harman, and R. M. Hierons, "How to Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation Using Co-evolution," in *GECCO (2)*. Springer, 2004, pp. 1338–1349.

[38] B. J. M. Grün, D. Schuler, and A. Zeller, "The Impact of Equivalent Mutants," in *ICST Workshops*, 2009, pp. 192–199.

[39] D. Schuler, V. Dallmeier, and A. Zeller, "Efficient Mutation Testing by Checking Invariant Violations," in *ISSTA*, 2009.

[40] D. Schuler and A. Zeller, "Covering and uncovering equivalent mutants," *Softw. Test. Verif. Rel.*, vol. 23, no. 5, pp. 353–374, 2013.

[41] ——, "(Un-)Covering Equivalent Mutants," in *ICST*, 2010, pp. 45–54.

[42] S. Nica and F. Wotawa, "Using constraints for equivalent mutant detection," in *WS-FMDS*, 2012, pp. 1–8.

[43] M. Kintis and N. Malevris, "Identifying more equivalent mutants via code similarity," in *Software Engineering Conference (APSEC, 2013 20th Asia-Pacific*, vol. 1, Dec 2013, pp. 180–188.

[44] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Jozala, "Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation," *IEEE Trans. Softw. Eng.*, vol. 40, no. 1, pp. 23–42, 2014.

[45] M. Kintis, M. Papadakis, and N. Malevris, "Isolating first order equivalent mutants via second order mutation," in *ICST*, 2012, pp. 701–710.

[46] ——, "Employing second-order mutation for isolating first-order equivalent mutants," *Softw. Test. Verif. Rel.*, pp. n/a–n/a, 2014.

[47] M. Papadakis, M. Delamaro, and Y. L. Traon, "Mitigating the effects of equivalent mutants with mutant classification strategies," *Sci. Comput. Program.*, vol. 95, pp. 298–319, 2014.

[48] M. Kintis and N. Malevris, "Using data flow patterns for equivalent mutant detection," in *ICST Workshops*, 2014, pp. 196–205.

[49] ——, "Medic: A static analysis framework for equivalent mutant identification," *Information and Software Technology*, vol. 68, pp. 1 – 17, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0950584915001329

[50] S. Bardin, M. Delahaye, R. David, N. Kosmatov, M. Papadakis, Y. L. Traon, and J. Marion, “Sound and quasi-complete detection of infeasible test requirements,” in *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*, 2015, pp. 1–10. [Online]. Available: http://dx.doi.org/10.1109/ICST.2015.7102607

[51] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, “An Experimental Determination of Sufficient Mutant Operators,” *ACM T. Softw. Eng. Meth.*, vol. 5, no. 2, pp. 99–118, April 1996.

[52] A. S. Namin, J. H. Andrews, and D. J. Murdoch, “Sufficient Mutation Operators for Measuring Test Effectiveness,” in *ICSE*, 2008, pp. 351–360.

[53] R. H. Untch, A. J. Offutt, and M. J. Harrold, “Mutation Analysis Using Mutant Schemata,” in *ISSTA*, 1993, pp. 139–148.

[54] M. Kintis, M. Papadakis, and N. Malevris, “Evaluating mutation testing alternatives: A collateral experiment,” in *APSEC*, 2010, pp. 300–309.

[55] P. Ammann, M. E. Delamaro, and J. Offutt, “Establishing theoretical minimal sets of mutants,” in *ICST*, 2014, pp. 21–30.

[56] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. Le Traon, “Threats to the validity of mutation-based test assessment,” in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016, pp. 354–365.

[57] B. Kurtz, P. Ammann, J. Offutt, M. E. Delamaro, M. Kurtz, and N. Gökçe, “Analyzing the validity of selective mutation with dominator mutants,” in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 571–582.

[58] G. Kaminski, P. Ammann, and J. Offutt, “Better predicate testing,” in *AST*, 2011, pp. 57–63.

[59] ——, “Improving logic-based testing,” *J. Syst. Software*, vol. 86, no. 8, pp. 2002–2012, 2013.

[60] R. Just, G. M. Kapfhammer, and F. Schweiggert, “Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis,” in *ISSRE*, 2012, pp. 11–20.

[61] K.-C. Tai, “Theory of fault-based predicate testing for computer programs,” *IEEE Trans. Softw. Eng.*, vol. 22, no. 8, pp. 552–562, 1996.

[62] W. B. Langdon, M. Harman, and Y. Jia, “Efficient multi-objective higher order mutation testing with genetic programming.” *J. Syst. Software*, pp. 2416–2430, 2010.

[63] M. Harman, Y. Jia, P. Reales Mateo, and M. Polo, “Angels and monsters: An empirical investigation of potential test effectiveness and efficiency improvement from strongly subsuming higher order mutation,” in *ASE*, 2014, pp. 397–408.

[64] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, “Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria,” *IEEE Trans. Softw. Eng.*, vol. 32, no. 8, pp. 608–624, 2006.

[65] Y. Ma, M. J. Harrold, and Y. R. Kwon, “Evaluation of mutation testing for object-oriented programs,” in *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, 2006, pp. 869–872. [Online]. Available: http://doi.acm.org/10.1145/1134437

[66] Y. Jia and M. Harman, “MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language,” in *TAIC PART*, 2008, pp. 94–98.

[67] P. Lam, E. Bodden, O. Lhotak, and L. Hendren, “The soot framework for java program analysis: a retrospective,” in *Cetus Users and Compiler Infastructure Workshop (CETUS 2011)*, 2011.

[68] M. Hutchins, H. Foster, T. Goradia, and T. J. Ostrand, “Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria,” in *Proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, May 16-21, 1994.*, 1994, pp. 191–200.

[69] M. E. Delamaro, L. Deng, V. H. S. Durelli, N. Li, and J. Offutt, “Experimental evaluation of sdl and one-op mutation for c,” in *ICST*, 2014, pp. 203–212.

[70] L. S. Ghandehari, J. Czerwonka, Y. Lei, S. Shafiee, R. Kacker, and D. R. Kuhn, “An empirical comparison of combinatorial and random testing.” in *ICST Workshops*, 2014, pp. 68–77.

[71] M. Patrick, R. Alexander, M. Oriol, and J. Clark, “Probability-based semantic interpretation of mutants,” in *Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on*, March 2014, pp. 186–195.

[72] L. Deng, J. Offutt, and N. Li, “Empirical evaluation of the statement deletion mutation operator,” in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, March 2013, pp. 84–93.

[73] L. Madeyski and N. Radyk, “Judy - a mutation testing tool for java,” *IET Software*, vol. 4, no. 1, pp. 32–42, 2010. [Online]. Available: http://dx.doi.org/10.1049/iet-sen.2008.0038

[74] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, and N. Malevris, “Analysing and comparing the effectiveness of mutation testing tools: A manual study,” in *16th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM’16)*, 2016, pp. 147–156.

[75] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, and B. Regnell, *Experimentation in Software Engineering*. Springer, 2012.