Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique

Mike Papadakis*, Yue Jia†, Mark Harman†, and Yves Le Traon*
*Interdisciplinary Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg
†CREST Centre, University College London, UK
michail.papadakis@uni.lu, mark.harman@ucl.ac.uk, yue.jia@ucl.ac.uk and yves.letraon@uni.lu

***Abstract*—Identifying equivalent mutants remains the largest impediment to the widespread uptake of mutation testing. Despite being researched for more than three decades, the problem remains. We propose Trivial Compiler Equivalence (TCE) a technique that exploits the use of readily available compiler technology to address this long-standing challenge. TCE is directly applicable to real-world programs and can imbue existing tools with the ability to detect equivalent mutants and a special form of useless mutants called duplicated mutants. We present a thorough empirical study using 6 large open source programs, several orders of magnitude larger than those used in previous work, and 18 benchmark programs with hand-analysis equivalent mutants. Our results reveal that, on large real-world programs, TCE can discard more than 7% and 21% of all the mutants as being equivalent and duplicated mutants respectively. A human-based equivalence verification reveals that TCE has the ability to detect approximately 30% of all the existing equivalent mutants.**

I. Introduction

Mutation testing has been shown to be a valuable testing technique, capable of simulating real faults [1], [2], [3] and almost every other testing adequacy criteria [4], [5], [6], [7]. Mutants are versions of the program under test in which a fault is deliberately inserted in order to either assess test effectiveness [8], [9], [10] or to support generation of effective tests [4], [11], [12], [13].

Despite its undisputed potential, the perception of mutation testing is that it is expensive. This is partly due to the large number of mutants and partly because of the 'equivalent mutant problem'; the topic of this paper. While the problem of the large number of mutants has largely been addressed in the literature by techniques such as mutant sampling [14], [15], [16] and the mutant execution optimisations [12], [17], [18], the equivalent mutant problem remains a challenge, particularly because the underlying scientific question is undecidable [19].

The problem with equivalent mutants is that we may make a syntactic change to a program, yet leave its semantics unaffected; some mutants may prove to be equivalent to the original program from which they are constructed. Unfortunately, the question of program equivalence, of course, undecidable, so we can never hope for a complete solution to the equivalent mutant problem.

If we can find fast scalable, and reasonably effective ways to reduce the incidence of equivalent mutants, we may sufficiently overcome the practical ramifications of equivalent mutants. The equivalent mutant problem is such a large stumbling block to mutation testing that many researchers believe that its effective removal would be sufficient to make mutation testing practical and widely applicable.

Our goal is to provide empirical evidence to support the claim that a straightforward and effective equivalent mutant detection technique already exists, yet it is not exploited. Recently, several mutation testing tools have been implemented for popular programming languages like C and Java [10]. However, *none* includes *any* technique for equivalent mutant detection.

Our results demonstrate that a simple, scalable and widely-applicable technique, which we call 'Trivial Compiler Equivalence (TCE)' can be exploited to imbue existing mutation tools and techniques with the ability to detect as many as 30% of equivalent mutants. We present results that evaluate TCE on benchmarks with known sets of equivalent mutants and also large-scale systems, several orders of magnitude larger than those used in any previous work on equivalent mutant detection. Our TCE approach has now been incorporated into the MiLu mutation testing tool, making it the first tool that supports equivalent mutant detection fully automated.

We present a thorough empirical study of TCE's potential to address this long-standing mutation testing challenge. Our results are surprising: the simple TCE approach can detect about 30% of all equivalent mutants thereby having the potential to dramatically save human effort in mutation testing. Since the technique is conservative, all identified equivalent mutants can safely be discarded.

TCE has other applications in mutation testing and beyond. For example, in mutation testing, there would be no point in including two mutants that are equivalent to each other, even if they are not equivalent to the original program from which they are constructed; either one or the other of these two mutually equivalent mutants can be discarded, saving some effort. We refer to these mutants as 'duplicated' mutants. This question of 'mutual mutant equivalence' has not been studied before. Our findings show that at least 21% of mutants are duplicated and can be discarded.

Our findings may also have implications beyond mutation testing. For example, in software development environments, it is quite often the case that developers make small changes to a system, for example to fix the bug (the inverse of mutate testing, which insert synthetic bugs). The question arises as to whether it would be worthwhile for the software development environment to include, as a sanity check, a check for equivalence using TCE. Our results suggest that this is possible, given the compilation time involved, but more importantly, our results show that it is also potentially useful; 7% of all simple edits (mutants) turn out to be TCE equivalent. Surely a developer would like to have this sanity check information available after each edit?

The rest of the paper is organized as follows: Section II presents mutation testing and related approaches. Section III details our experiment and the studied research questions, while, Section IV analyses our results. Our findings are discussed in Section V. Finally, the threats to validity and our conclusions are presented in Sections VI and VII.

II. Background

A. 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 current version of MiLu.

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 called as *killed*, while in the opposite case it is called as *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.

Undecidability of equivalencies 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 [20]. 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.

B. Equivalent Mutants

Early research on mutation testing has demonstrated that deciding whether a mutant is equivalent is an undecidable problem [19]. Fortunately, partial and heuristic solutions exist [21]. However, tackling the equivalent mutant problem is hard. This is evident by the fact that very few attempts exist. In literature this problem is tackled in two ways. One is to address the problem directly by detecting some equivalent mutants, while, the second one is to avoid them by identify likely non-equivalent ones or help with the manual analysis. We refer to them as the *Detect* and *Reduce* approaches, respectively.

Table I summarizes the current state-of-the-art techniques in chronological order. From this table it becomes evident that very few methods and tools exist. Regarding the equivalent mutant detection, only one publicly available tool exist with the largest considered subject being composed of 29 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 components/classes that these mutants are located. In these lines, in Table I we report the size of the projects that we consider.

Acree [23] studied 25 killable and 25 equivalent mutants, and found that testers correctly identified equivalent mutants for approximately 80% of the cases. In 12% of the cases, they identified equivalent mutants as killable, while, in 8% of the cases they identified killable mutants as equivalent. Therefore, indicating that detection techniques, such as the one suggested by the present paper, not only help at saving resources but also at reducing the mistakes made by the humans.

The idea of using compiler optimization techniques to detect equivalent mutants was suggested by Baldwin and Sayward [22]. The main intuition behind this technique is that code optimization rules, such as those implemented by compilers, form transformations on equivalent programs. Thus, when the original program can be transformed by an optimization rule to one of its mutants, then, this mutant is, *ipso facto*, equivalent. Baldwin and Sayward proposed adapting six compiler optimization transformations. These transformations were then studied by Offutt and Craft [21] 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. However, we propose a truly simple (and therefore scalable and directly exploitable) use of compilers, which remained unexplored. Our TCE simply declares equivalencies only for those mutants which their compiled object code is identical to the compiled object code of the original program.

Offutt and Pan [24], [25] 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.

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

Author(s) [Reference]	Year	Language	Largest Subject	#Eq. Mutants	Available Tool	Category	Findings
Baldwin & Sayward [22]	1979	-	-	-	-	**Detect**	Compiler optimization can be used to detect eq. mutants
Acree [23]	1980	Fortran	-	25	-	**Detect**	Humans make mistakes when they identify eq. mutants
Offutt & Craft [21]	1994	Fortran	52	255	-	**Detect**	Compiler optimization can detect on average 45% of eq. mutants
Offutt & Pan [24], [25]	1996-7	Fortran	29	695	✓	**Detect**	Constraint-based testing can detect on average 47% of eq. mutants
Voas & McGraw [26]	1997	-	-	-	-	**Detect**	Slicing may be helpful in detecting eq. mutants
Hierons *et al.* [27]	1999	-	-	-	-	**Detect** /Reduce	Program slicing can be used to detect and assist the identification of eq. mutants
Harman *et al.* [28]	2001	-	-	-	-	**Detect** /Reduce	Dependence analysis can be used to detect and assist the identification of eq. mutants
Adamopoulos *et al.* [29]	2004	-	-	-	-	Reduce	Co-evolution can help in reducing the effects of eq. mutants
Grun *et al.* [30]	2009	Java	12,449	8	✓	Reduce	Coverage Impact can be used to classify killable mutants
Schuler *et al.* [31]	2009	Java	94,902	10	✓	Reduce	Invariants violations can be used to classify killable mutants
Schuler & Zeller [32], [33]	2010-2	Java	94,902	63	✓	Reduce	Coverage Impact can be used to classify killable mutants
Nica & Wotawa [34]	2012	Java	380	1424	-	**Detect**	Constraint-based testing can detect eq. mutants
Kintis *et al.* [35], [36]	2012-4	Java	94,902	89	-	Reduce	Higher order mutants can be used to classify killable mutants
Kintis & Malevris [37]	2014	Java	25,909	84	-	**Detect**	Data-flow patterns can detect 69% of the eq. mutants introduced by the AOIS operator
Papadakis *et al.* [38]	2014	C	513	5,589	-	Reduce	Coverage Impact can be used to classify killable mutants
This paper	**2014**	**C**	**362,769**	**9,551**	✓	**Detect**	Compilers can be used to effectively automate the mutant equivalence detection

Nica and Wotawa [34] implemented a similar constraint-based approach to detect equivalent mutants and demonstrated that many equivalent mutants can be detected. Voas and McGraw [26] suggested that program slicing can help in detecting equivalent mutants. Later, Hierons *et al.* [27] 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, [24], [25], 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 [37] used data-flow patterns to detect equivalent mutants. They showed that 69% of the equivalent mutants produced by the AOIS operator, i.e., insertion of the increment/decrement operator, can be detected. Since, this approach works only on one operator, lacks implementation and its evaluation was based on 84 instances, it leaves the practicality and scalability questions open.

Hierons *et al.* [27] suggested using program slicing to reduce the size of the program considered during the equivalence identification. Thus, tester can focus on the code relevant to the examined mutants. Harman *et al.* [28] also suggested using dependence-based analysis as a complementary method to assist in the detection of equivalent mutants.

Adamopoulos *et al.* [29] 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.* [30] and developed by the studies of Schuler *et al.* [31] and Schuler and Zeller [32], [33] who found that impact on coverage can accurately classify killable mutants. Papadakis *et al.* [38] proposed a mutation testing strategy that takes advantage of mutant classification. Kintis *et al.* [35], [36] further develop the approach, using the impact of mutants on other mutants, i.e., using higher order mutants.

C. 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 [15] can be used as cost effective alternatives to perform mutation testing [16] and [14]. 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.* [39] demonstrated that five mutant operators are almost as effective as the whole set of operators.

More recently, Namin *et al.* [40] used statistically identified optimal operator subsets. Other cost reduction methods involve mutant schemata [41], [18], [12]. This technique works by parameterizing all the mutants through instrumentation, i.e., introduce all the mutants into one parameterized program. However, apart from the inherent limitations of this technique [42] 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.* [43] defined the notion of disjoint mutants, i.e., a set of mutants that subsumes all the others, and found 9% to subsume all. Ammann *et al.* defined minimal mutants [44] showing that this may reduce the number of mutants. Kaminski *et al.* [45], [46] and Just *et al.* [47] leverage the suggestions made by Tai [48] 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 [16], [49] and searching [50], [51], [52] within the space of higher order mutants both reduce the number of mutants and also of the equivalent mutants.

III. Experimental Study and Settings

This section details the settings of our experiment. It presents the TCE approach, in III-A, our research questions, in III-B, the programs used, in III-C, the employed mutant operators, in III-D and the execution environment, in III-E.

A. Detecting Mutant Equivalencies, the TCE approach

Executable program generation involves several transformation phases that change the machine code. Different optimization transformation techniques result in different executables. However, when we have multiple program versions with identical source code, then there is no point differentiating them with test data; it is safe to declare them as functionally equivalent. TCE realizes this idea to detect mutant equivalencies. 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 readily appreciate TCE implementation is truly trivial, hence the name: it is a compile command combined with a comparison of binaries.

B. 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 at detecting equivalent and duplicated mutants. This poses our first RQ:

RQ1 (Effectiveness): How effective is the TCE approach at 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`.

To reduce the confounding effects of different compiler configurations, we apply four popular options and report the number of 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.

Existing literature on mutant equivalent detection techniques suffers 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 some empirical study to determine the degree to which it scales. Hence, in our 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 six large open source projects and we report the efficiency of the mutant generation, equivalent mutant detection and duplicated mutant detection processes. We also explored 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 equivalent mutants. However, to what extent? This is investigated by our 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 a benchmark set [20] with hand-analysed ground-truth data on equivalent mutants. The benchmark includes 990 manually identified equivalent mutants over 18 small and medium sized subjects.

We report the proportion of the equivalent mutants found by the 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 the TCE approach is useful in the sense that mutation testers can better understand the importance of their choices. Hence, our next RQ is to examine the extent of the equivalent and duplicated mutants found per mutant operator:

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

Finally, we investigate whether the size of the programs or the number of mutants they contain correlates with the effectiveness of the TCE approach. One might expect that in larger programs, the equivalent mutant identification would be harder, thereby impeding the TCE's effectiveness.

RQ5 (Correlation Analysis): Does program size or number of mutants affect TCE?

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

C. Subject Programs

We used two sets of subjects. The first is composed of six large open source programs. In this set, we choose 'real-world' programs that vary in size and application domain. The second set was taken from the study of Yao *et al.* [20] and it is composed of 18 subjects. We choose this set because it is accompanied by manually identified equivalent mutants. The availability of known equivalent mutants allows us to answer the RQ3, because it provides a 'ground truth' on the undersidable 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 IV-B, 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, we rank their source files according to their lines of code. Then, we select the two largest components. On these two components we apply mutation to all the functions they contain.

Table II presents the information about the first set of subjects. The second set contains 8 programs with 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 this programs can be found in [20].

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.

TABLE II: Subject program details: the 'LoC' shows the lines of code of the project; 'Comp' and 'Comp-Size' shows the components considered and their size; The '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	tree	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**

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 for *MSMTP*.

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.

D. Mutant Operators

Based on previous research on mutant operator selection, we identify and use two sets of operators.The first set, proposed by Offutt *et al.* [39], is composed of five operators, i.e., ABS, AOR, LCR, ROR, and UOI. We use this set due to its extensive use in literature [10]. The second set was used in the studies of Andrews *et al.* [2], [53], where it was shown that it provides accurate predictions of the real fault detection ability of the test suites.

This set is composed of eight operators; three of them, i.e., AOR, LCR and ROR, are drawn from the first set, while, the other five, i.e., CRCR, OAAA, OBBN, OCNG and SSDL, are designed to cater the common C faults. A detailed description of the operators is reported in Table III.

To generate the mutants, we use MILU [54], an open source mutation testing tool for C. We detail exactly how the 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 on numerical variables. The CRCR was applied to integer and floating numeric constants.

TABLE III: The mutant operators used.

Name	Description
ABS: *Absolute Value Insertion*	$\{(e,\texttt{abs}(e)),(e,\texttt{-abs}(e))\}$
AOR: *Arithmetic Operator Replacement*	$\{(x,y) \mid x,y \in \{+,-,\star,/,\backslash\%\} \wedge x \neq y\}$
LCR: *Logical Connector Replacement*	$\{(x,y) \mid x,y \in \{\&\&,\mid\mid\} \wedge x \neq y\}$
ROR: *Relational Operator Replacement*	$\{(x,y) \mid x,y \in \{>,>=,<,<=,==,!=\} \wedge x \neq y\}$
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*	$\{(x,y) \mid x,y \in \{+=,-=,\star=,/=,\backslash=\%\} \wedge x \neq y\}$
OBBN: *Bitwise operator mutation*	$\{(x,y) \mid x,y \in \{\&,\mid\} \wedge x \neq y\}$
OCNG: *Logical context negation*	$\{(e,!(e)) \mid e \in \{\texttt{if(e)},\texttt{while(e)}\}\}$
SSDL: *Statement Deletion*	$\{(s,\texttt{remove}(s))\}$

No mutant operator was applied on the variables of the lefthand side of assignment statements; we only apply them at the right hand sides. All operators are applied recursively to all sub expressions. Further details and the implementation of the operators can be found on the webpage of MILU[1].

E. Experimental Environment

All our experiments were undertaken on the Microsoft Azure Cloud platform using a A9 Compute Intensive Instance in the Ubuntu 14.04 operating system with compiler `gcc 4.8`. 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 realized 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'.

IV. Results & Answers to Research Questions

This section reports our results and provides answers to the research questions.

A. RQ1: 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.

[1]https://github.com/yuejia/Milu/tree/develop/src/mutators

Table IV reports our results per program and per considered optimization option. Overall, these results indicate that TCE can detects 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 remove approximately 28% of all mutants.

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 3%, while, in the best it is 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 IV reveals that, depending on the options used, the detected equivalencies 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 optimization options, i.e., *-O*, *-O2* and *-O3*, there is no clear winner and their behavior varies between programs. However, the overall differences are relatively small between the options. With respect to duplicated mutants, the results are clear and they show that the best options are the *-O2* and *-O3*.

B. RQ2: TCE Efficiency

To assess the efficiency of the TCE approach and answer the RQ2, we report the CPU execution time. Table V 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.

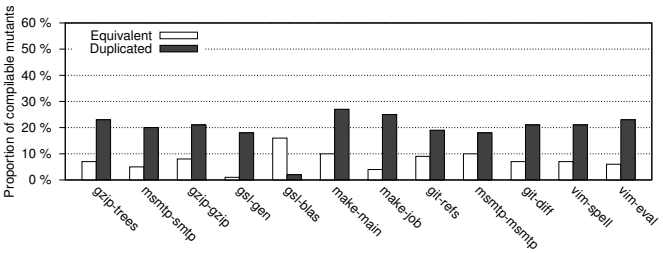

Fig. 1: The proportion of equivalent and duplicated mutants detected by TCE per program studied.

TABLE IV: Equivalent and duplicated mutants detected by TCE. 'None', '-O', '-O2' and '-O3' report the fraction of all identified equivalent mutants that were detected per optimization flag. 'Number of 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		Number of 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 V: Execution time: 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**

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, on all the 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 equivalencies only 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 respectively 374,162, 260 and 2,744 seconds. Therefore, TCE analyzed 129,161 mutants in 377,166 seconds. This time accounts for less than 3 seconds per mutant suggesting that it is reasonably fast.

C. RQ3: 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.* [20], using the accompanying website data from www0.cs.ucl.ac.uk/staff/Y.Jia/projects/equivalent_mutants/. This site is regularly updated, so data may differ slightly from those previously reported [20]. Additional details about these data can be found on the website.

TABLE VI: Number 'No.' and proportion '%' of detected equivalent mutants on the Yao *et al.* [20] benchmark set.

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%**

Table VI 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 TCE can detect from 9% to 100% of all the equivalent mutants. This number accounts for 30% on average which is approximately 7% of all the mutants (both killable and equivalent mutants). 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 the a), b), and c), respectively.

D. RQ4: 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 VII reports the number and proportions of the equivalent and duplicated mutants found by TCE per program 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 VII, 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.

E. RQ5: Program Size and Mutant Equivalencies

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).

We found a correlation between the number of mutants and the number of equivalent mutants detected ($\rho = 0.818$). Since more mutants leads to more equivalent ones [20], this result suggests that TCE can identify a certain amount of them. A stronger correlation was also found for the number of mutants and the detected duplicated ($\rho = 0.930$). This is a surprising result since it indicates that a certain percentage of duplicated mutants is always produced by the operators.

We also study the relation between the program size and the number of detected equivalent mutants. We found a medium to small correlation ($\rho = 0.692$). A slightly lower correlation was found between the size of program and the number of duplicated mutants ($\rho = 0.650$).

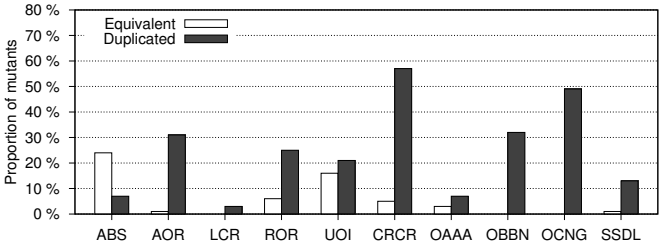

Fig. 2: The proportion of equivalent and duplicated mutants detected by TCE per mutant operator

TABLE VII: 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%**

We found a medium correlation between the size of program and the whole number of mutants ($\rho = 0.671$). This result is surprising since it suggests that larger programs do not necessarily involve a higher number of mutants. In conclusion, our results indicate that both the number of mutants and the equivalences detected by TCE are more related to the code characteristics of the programs than to their size.

V. Implications

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

We have seen that TCE reduces the total number of mutants by 28%, where 7% are equivalent and 21% are duplicated. The existence of so many 'useless' mutants has a distorting influence on the accuracy of the mutation score measurement. According to our results, at least the 21% of all the mutants are duplicated. This implies that the true mutation score might be underestimated or overestimated by killed or live duplicated mutants.

Therefore, it is possible that when we compare testing methods, one might have a 'fake' advantage (by killing more duplicated mutants than the other). Future empirical studies must remove these mutants to avoid biased results.

The time to detect equivalent and duplicated mutants is approximately 22 and 225 seconds per program. This indicates that once the mutants have been compiled, 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.

Interestingly, the detected mutant equivalencies 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 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 we cannot ignore it. All previous work seems to assume 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 using the four most popular compiler settings.

We explored the main `gcc` settings covering a wide range of optimization options and found that all of them can be used to detect mutant equivalencies (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* option 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. We also studied only first order mutants. However, TCE is generic and can be applied to detect equivalent and duplicated higher order mutants.

Finally, we can advance our approach by using some form of either binary abstraction such as semantic interpretation, as done by BinJuice [55], or checksum, as done by md5. The fist case might have an effect on the effectiveness of TCE while, the second one on the performance of the diff comparisons.

VI. Threats to validity

As it is usual in software engineering experiments, our subjects might not be representative. To ameliorate this issue, we selected 6 real-world programs, several orders of magnitude larger than those used in previous equivalent mutant detection studies, of varying size and application domain. We also performed an additional evaluation using a different set of programs, composed of 18 benchmark subjects, taken from the literature. Our approach provided similar results in these two sets, i.e., on average it detects equivalent mutants approximately as 7.4% of all the mutants on the 6 real-world programs, while on the 18 subjects of the literature it detects approximately 7.2%. Additionally, our results are in line with those reported in literature[2] providing confidence that they are realistic. We studied the mutants of the C language and TCE implemented using `gcc`. Additional studies are needed to determine TCE performance on other languages.

Other threats are due to the use of software systems. Thus, the `gcc` compiler and the 'diff' utility may have defects. However, these systems are heavily tested and deployed. Thus, it is unlikely that they would have defects that would influence our results to a great extent. Implementation defects of MiLu may have an influence. To reduce this threat we carefully check its results. However, we consider this threat as small since MiLu has also been used by several authors in their studies, e.g., [4], [57], independently of us. Furthermore, we used the Yao *et al.* [20] benchmark, which was entirely built by hand. These results served as a 'sanity check' to reduce the threat to validity.

[2]Offutt and Pan [25] reported that 9% of all the mutants are equivalent. Delamaro *et al.* [56] found 12% , Kintis *et al.*, Schuler and Zeller [36], [32] 7%-8%, Papadakis *et al.* [38] 17% and Yao *et al.* [20] 23%. Thus, the actual ratio of the equivalent mutants is in the range of 8%-25%. TCE detects 7.4% which is more or less 30%, similar to what we found in RQ3.

Our results might be affected by our choice of mutation operators. To ameliorate this threat, we used all the popular operators (included in most existing mutation testing tools). We also included those empirically found to correlate with fault detection.

The use of the Yao *et al.* [20] benchmark may also poses another threat. This is due to the manual analysis performed: some killable mutants may have been mistakenly identified as equivalent. However, this study was performed independently of the present one and hence, it is not likely that these kind of mistakes coincidentally match the results of TCE. Additionally, it is equally possible that any such mistakes have also led to the effectiveness of our method being underestimated.

Finally, all our subjects, tools and data are available in the accompanied website of the present paper[3]. This helps reducing all the above-mentioned threats [58] since independent researchers can check, replicate and analyse our findings.

VII. Conclusions

Software engineering is a unique mix of both science and engineering. Sometimes researchers (including the authors of this paper) can become so bedazzled by their search for clever science, that they may overlook simple yet effective engineering solutions that lie within their grasp. We report on one such case in this paper: the search for techniques to ameliorate the effect of equivalent mutants on mutation testing.

Previous, work has concentrated on complicated detection techniques involving sophisticated semantic dependence analysis, meaning-preserving transformation rules and constraint solving techniques. Unfortunately, these techniques proved neither sufficiently scalable nor widely applicable to have become adopted in production tools or research prototypes.

By contrast, the Trivial Compiler Equivalence approach that we advocate in this paper is both as scalable and as widely applicable as the compiler technology upon which it rests. We simply declare to be equivalent, any mutant for which the compiled object code is identical to the program from which is constructed. Our contribution does not lie in the proposal of this technique, which is, as the name implies, *trivial*. Rather, the contribution of this paper is to demonstrate that TCE can detect a large proportion of real-world equivalent mutants.

Our empirical study on 18 benchmarks shows that 30% of equivalent mutants can be detected by TCE. Perhaps more importantly, in a set of six large programs we found that TCE can also detect more than 7% of all mutants to be equivalent; perhaps as many as 30% of all equivalent mutants respectively.

VIII. Acknowledgments

Mike Papadakis is supported by the National Research Fund, Luxembourg, INTER/MOBILITY/14/7562175. 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.

[3] http://www0.cs.ucl.ac.uk/staff/Y.Jia/projects/compiler_equivalence/

REFERENCES

[1] 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.

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

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

[4] 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.

[5] 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.

[6] 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.

[7] P. G. Frankl, S. N. Weiss, and C. Hu, "All-Uses Versus Mutation Testing: An Experimental Comparison of Effectiveness," Polytechnic University, Brooklyn, New York, Technique Report, 1994.

[8] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness." in *ICSE*, 2014, pp. 435–445.

[9] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov, "Comparing non-adequate test suites using coverage criteria," in *ISSTA*, 2013, pp. 302–313.

[10] 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.

[11] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," in *ISSTA*, 2010, pp. 147–158.

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

[13] M. Harman, Y. Jia, and W. B. Langdon, "Strong higher order mutation-based test data generation," in *FSE*, 2011, pp. 212–222.

[14] 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.

[15] 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.

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

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

[18] R. Just, M. D. Ernst, and G. Fraser, "Efficient mutation analysis by propagating and partitioning infected execution states," in *ISSTA*, 2014, pp. 315–326.

[19] 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.

[20] 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.

[21] 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.

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

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

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

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

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

[27] 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.

[28] 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.

[29] 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.

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

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

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

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

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

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

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

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

[38] 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.

[39] 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.

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

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

[42] 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.

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

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

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

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

[47] 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.

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

[49] 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.

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

[51] 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.

[52] 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.

[53] 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.

[54] 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.

[55] A. Lakhotia, M. D. Preda, and R. Giacobazzi, "Fast location of similar code fragments using semantic 'juice'," in *PPREW@POPL*, 2013, p. 5.

[56] 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.

[57] 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.

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