Evaluation of Mutation Testing in a Nuclear Industry Case Study

Pedro Delgado-Pérez, Ibrahim Habli, Steve Gregory, Rob Alexander, John Clark and Inmaculada Medina-Bulo

***Abstract*—For software quality assurance, many safety-critical industries appeal to the use of dynamic testing and structural coverage criteria. However, there are reasons to doubt the adequacy of such practices. Mutation testing has been suggested as an alternative or complementary approach but its cost has traditionally hindered its adoption by industry, and there are limited studies applying it to real safety-critical code. This paper evaluates the effectiveness of state-of-the-art mutation testing on safety-critical code from within the UK nuclear industry, in terms of revealing flaws in test suites that already meet the structural coverage criteria recommended by relevant safety standards. It also assesses the practical feasibility of implementing such mutation testing in a real setting. We applied a conventional selective mutation approach to a C codebase supplied by a nuclear industry partner and measured the mutation score achieved by the existing test suite. We repeated the experiment using trivial compiler equivalence (TCE) to assess the benefit that it might provide. Using a conventional approach, it first appeared that the existing test suite only killed 82% of the mutants, but applying TCE revealed that it killed 92%. The difference was due to equivalent or duplicate mutants that TCE eliminated. We then added new tests to kill all the surviving mutants, increasing the test suite size by 18% in the process. In conclusion, mutation testing can potentially improve fault detection compared to structural-coverage-guided testing, and may be affordable in a nuclear industry context. The industry feedback on our results was positive, although further evidence is needed from application of mutation testing to software with known real faults.**

***Index Terms*— Mutation testing; safety-critical systems; coverage criteria; verification and validation; nuclear industry.**

I. Introduction

MUTATION TESTING is a method for estimating the robustness of test suites by measuring their effectiveness for finding faults which have been systematically seeded in the code. Several faulty versions of the program under test (mutants) are generated, each one with a simple syntactic change, and the test suite is run against each faulty version. If the test results differ (typically, if some tests fail) when run against a faulty version, that version is said to be "killed". The effectiveness of the suite is the "mutation adequacy score" – the proportion of faulty versions that are correctly detected by the tests. Originally proposed in the 1970s by Hamlet [1] and DeMillo et al. [2], this technique has been widely studied by researchers [3] but has not been embraced by industry, which has regarded the cost as a millstone for its practical application.

Research studies on mutation testing have produced evidence of its usefulness in improving the quality of test suites and have also explored multiple mechanisms to reduce the cost without significantly lessening its effectiveness [4]. As a consequence, it has re-emerged as a feasible opportunity to enhance test assurance models adopted in some critical domains, where verification and validation is a key phase in software development. The potential for increasing confidence in existing test suites is especially appealing for safety-critical industries.

In safety-critical industries such as aviation, automotive and nuclear, where failures of certain software-based functions may lead to human harm or damage to the environment, system developers and integrators need evidence from rigorous testing to meet regulatory requirements. The form and level of rigour vary, but is most often expressed as a specific structural coverage criterion. For example, the safety standards *IEC 61508* [5], *ISO 26262* [6] and *DO 178C* [7] adopt this approach. The degree of rigour and coverage required — e.g., statement coverage, branch coverage, or Modified Condition/Decision Coverage (MC/DC) [8] — depends on the criticality of the software to safety. For example, *DO 178C* requires MC/DC at the highest level of software assurance (e.g., that applying to aircraft engine controllers).

This study aims to provide rigorous and empirical evidence of the impact that the application of mutation testing could have in a nuclear industrial setting. Our top-level research question is as follows:

Submission date: 27/07/2017

This study was funded by the UK Control and Instrumentation Nuclear Industry Forum (CINIF). This paper was also partially funded by the European Commission (FEDER) and the Spanish Ministry of Economy and Competitiveness (National Program for Research, Development and Innovation) through the project DArDOS (TIN2015-65845-C3-3-R).

Pedro Delgado-Pérez is with the Department of Computer Science and Engineering, University of Cádiz, Cádiz, Spain. (e-mail: pedro.delgado@uca.es).

Ibrahim Habli is with the Department of Computer Science, University of York, UK (e-mail: ibrahim.habli@york.ac.uk).

Steve Gregory is with AWE, Aldermaston, Reading, UK (e-mail: steve.gregory@awe.co.uk).

Rob Alexander is with the Department of Computer Science, University of York, UK (e-mail: rob.alexander@york.ac.uk).

John Clark was with the Department of Computer Science, University of York, UK. He is now with Department of Computer Science, University of Sheffield, Sheffield, UK (e-mail: john.clark@sheffield.ac.uk).

Inmaculada Medina-Bulo is with the Department of Computer Science and Engineering, University of Cádiz, Cádiz, Spain. (e-mail: inmaculada.medina@uca.es).

Can mutation testing affordably enhance fault detection in safety-critical systems that are assessed as suitable for use in nuclear safety applications?

In this paper, we explore this question using a case study, supported by a series of experiments, of a safety-related software system and its associated branch-adequate test suites, which have been selected and evaluated in collaboration with partners from the UK nuclear industry. We evaluated mutation testing by applying two cost reduction techniques, namely (1) *selective mutation*, a well-known strategy that discards a subset of mutation operators, and (2) *trivial compiler equivalence*, a novel method to identify some equivalent and duplicate mutants (i.e., variants which are not useful for the assessment and refinement of the test suite). The results show that this approach can make mutation testing affordable for the industry while retaining testing power. The results can be summarised as follows:

- The selective set of mutation operators generates some mutants which are not killed by the existing branch-adequate test suites in most of the functions analysed.
- By adding new test cases to kill surviving mutants, the minimal size of the test suite is notably increased.
- Thanks to trivial compiler equivalence, we can detect a significant percentage of ineffective mutants automatically and therefore calculate the mutation score more accurately.

II. Background

A. Mutation Testing

1) *Overview*

Mutation testing is a technique used to evaluate the ability of a test suite in revealing faults in the source code [9]. In this technique, new versions of the program under test are generated. These versions are known as mutants, because they contain an intentionally injected fault. Mutation testing is founded on two underlying hypotheses: the *Competent Programmer Hypothesis* and the *Coupling Effect Hypothesis* [2].

The Competent Programmer Hypothesis suggests that programmers create programs that are very close to the correct version but may contain subtle, low-level faults. The simple syntactic changes introduced in mutation testing represent common programming mistakes.

The Coupling Effect Hypothesis suggests that complex faults are realised when simple faults combine and result in new behaviours. According to Offutt [10], "*complex faults are coupled to simple faults in such a way that a test data set that detects all simple faults in a program will detect a high percentage of the complex faults*".

The rationale behind mutation testing is that test suites that are deemed adequate by software engineers should be able to detect changes that are introduced into the code. As such, mutation testing provides an empirical test of the engineer's confidence in the test suite. The analysis of the mutants can also assist in improving the rate of fault detection of the test suite.

There are three main stages when applying mutation testing:

- *Mutant generation*: In this stage, the source code is analysed with respect to a set of mutation operators (syntactic transformations of the code) to determine where in the code mutations can be injected. For each location detected in the code, a mutant is generated. Each mutant is usually a clone of the original program except for a simple syntactic change.
- *Test suite execution*: Once the mutants have been produced, the original test suite is executed against each mutant to produce an output.
- *Mutant analysis*: The mutants are then classified as killed or alive depending on whether the test suite could detect the mutation (i.e., because of a difference in the output when compared to the original program) or not (because of no observable difference in the output when compared to the original program).

Mutants can be generated manually according to predefined mutation operators and the execution of the test suite can be prepared for each of those mutants. However, this is a laborious and error-prone task. Multiple tools have therefore been developed to undertake the two first stages systematically (see [3] for a survey).

The analysis of test execution results, however, is hard to fully automate. Ideally, the test suite should be able to detect all the mutations injected into the code and no further actions would be required because the test suite achieves full mutation coverage. In practice, there are usually some mutants undetected by the test suite. In that case, the tester needs to review those surviving mutants. The behaviour of some of the mutants may be the same as the original code (equivalent mutants), and therefore no test can detect the mutation. For instance, the fragment "*if (x > 1) x = 1;*" is behaviourally equivalent to "*if (x >= 1) x = 1;*" – whatever the value assigned to the variable *x* before the execution of this fragment, the variable will have the same value in both versions after the conditional statement. Where this is not the case, the test suite has indeed failed in detecting injected faults within the code.

Once all surviving mutants have been inspected and equivalent mutants have been discarded, the tester can measure the ability of the test suite to detect faults. The *mutation adequacy score* is the number of killed mutants divided by the number of non-equivalent mutants. The higher the mutation score, the higher the test suite quality and therefore its ability to reveal coding errors. The test suite is *mutant adequate* when the mutation score is 100%, that is, when it has killed the full set of non-equivalent mutants.

As an extra step, the engineer can create new test cases to kill the undetected non-equivalent mutants. The mutation testing process should then be repeated with the augmented test suite to ensure that the mutation score increases accordingly.

Mutation testing is a powerful technique but computationally inefficient in its basic form. There are two main problems when applying mutation testing: the high computational cost when generating and executing all the mutants, and the presence of equivalent mutants (determining which of the live mutants are equivalent requires manual inspection and takes considerable time). These problems are explored in the next section.

2) *Improving the Efficiency of Mutation Testing*

Mutation testing can generate a large number of mutants even for small programs, so most recent research in mutation testing has aimed to reduce the effort of applying it. Several techniques have been proposed to address this issue [4]. Some of them can be classified as “do fewer” techniques, in the sense that they seek to reduce the number of mutants, such as *random mutant selection* [11] (i.e., sampling a percentage of the full set of mutants) and *higher order mutation* [12] (i.e., several mutations are combined into a single mutant). *Selective mutation*, perhaps the cost reduction technique with the greatest acceptance [13], [14], [15], [16], works under the assumption that some mutation operators can be excluded without sacrificing a great deal of fault-revealing power. Notably, Offutt et al. [13] found that five of the 22 mutation operators implemented in the mutation tool *Mothra* sufficed to apply mutation testing in an effective way, allowing for large reductions in the number of mutants (78% on average).

For programs developed in C, which is the language of the software system examined in this paper, Barbosa et al. [14] defined a set of general rules to systematically select a subset of mutation operators. By applying this guideline to Agrawal et al.’s operators [17], the authors found that with only 10 operators the mutation score was still close to 100% (99.6% on average with 27 programs). Namin et al. [15] also tried to find sufficient sets of operators for C programs by defining a statistical analysis procedure to predict an effective subset of operators. The results show that using just 28 out of 108 operators leads to a good approximation of the full-set mutation score. One of the most recent studies on selective mutation was conducted by Delamaro et al. [16]. They used a greedy algorithm which successively added the operators that increased the overall mutation score the most. Unlike the previous two studies, the authors of that paper assessed mutation operators not only regarding the effectiveness but also considering the cost in the form of number of mutants and number of equivalent mutants.

While some techniques have been investigated to reduce and detect equivalent mutants [18], [19], [20], this is still an undecidable problem. Mutant classification strategies analysing coverage impact of mutations [21], [22] have been used in a study to mitigate the effects of equivalence. Papadakis et al. [23] also proposed a technique based on compiler optimisations (TCE) to automatically detect equivalent mutants. In their study, TCE was able to remove 30% of all existing equivalent mutants on average in 18 benchmark programs, and 7% (equivalent) and 21% (duplicate) of all mutants on 6 large open-source programs.

Another related work using C programs was conducted by Amman et al. [24]. They proposed to minimise the set of mutants to avoid the impact of redundant mutants when interpreting the mutation score. Based on this theoretical framework, they analysed the mutants generated with the mutation tool *Proteum* when applied to the *Siemens* suite, showing that the mutation scores were lower once redundant mutants were removed.

3) *Mutation Testing and Structural Test Coverage*

Andrews et al. [25] applied mutation testing to evaluate four test coverage criteria: block, decision, c-use and p-use. They showed that mutation testing can help in predicting the effectiveness of these criteria to detect real faults and their relative cost in terms of fault detection, test suite size and control/data flow coverage.

Yao et al. [26] showed the distribution of “stubborn” mutants across mutation operators. These authors labelled as stubborn those non-equivalent mutants that are not detected by a test suite complying with branch coverage criteria. They concluded that testers should prioritise those operators generating many of these mutants in comparison with the number of equivalent mutants.

Inozemtseva et al. [27] studied the correlation between coverage (statement, decision and MC/DC), test suite size and effectiveness of large programs. The results gave evidence that test effectiveness is not strongly correlated with coverage criteria, so coverage is not necessarily a good indicator of test quality.

4) *Significance of Mutation Testing for the Industry*

There has been some empirical evaluation of mutation testing in real testing environments. Daran and Thévenod-Fosse [28] also considered safety-critical software in a previous study, but with the aim of identifying whether mutations are correlated with real faults instead of evaluating the test suites developed. That study found a relation between mutations and real coding errors in a program from the civil nuclear field. Concretely, 85% of the injected mutations were also produced by real faults.

Andrews et al. [29] applied four mutant types in C to explore the link between hand-seeded and real faults. The results suggest that manually-seeded mutations are different from real faults and harder to detect, whereas mutation operators are more in line with real faults. The experiments by Just et al. [30] provide some evidence that the simple errors simulated by mutations relate to complex errors, supporting the coupling effect hypothesis. However, the results obtained by Gopinath et al. [31] contradict that hypothesis because real faults appeared to be more complex than most of the mutant types considered in that study.

Baker and Habli [32] carried out an empirical evaluation based on two safety-critical airborne systems that had satisfied the coverage requirements for certification. Those systems were developed using high-integrity subsets for C (MISRA C [33]) and Ada. In their experiments, they found an effective subset of mutation operators that was able to detect different deficiencies in tests suites which had already met statement and MC/DC coverage and had been manually peer-reviewed.

B. *UK Nuclear Industry*

The nuclear industry provides safety-critical services and develops technologies whose failure, under certain conditions, can lead to catastrophic events, i.e., resulting in harm to humans and damage to property and the environment. As such, it is a highly regulated domain with rigorous assessment practices. Requirements include a high degree of redundancy and diversity in design, both for hardware and software, and independence in testing and certification.

In the UK, the industry adopts a two-legged approach to the assurance of software-based systems, in accordance with the Office for Nuclear Regulation's (ONR) Safety Assessment Principles [34]. These two legs are "production excellence" and "confidence building" and the rigour applied is commensurate with the claimed risk reduction. For a software-based component to be incorporated into a safety-critical system it must first be qualified following the two-legged approach.

For Commercial-Off-The-Shelf (COTS) software-based instrumentation (smart devices), the manufacturer's development process and documentation are assessed, including all records of software analysis and testing. This assessment of production excellence is performed against the safety standard *IEC 61508* [5] as a benchmark, which requires structural test coverage criteria such as branch coverage and MC/DC. *IEC 61508* is applied along with some more stringent, industry-specific requirements. Independent analysis and testing are then carried out, usually by an expert and independent third party, using diverse tools and techniques. Provided that the results of these activities are favourable, the device may be deemed suitable for use.

Despite the robustness of the current approach to assuring software in the nuclear industry, there has been a general trend toward greater automation, particularly for instrumentation and control, potentially giving more authority to software-based functions [35]. To this end, mutation testing, in the context of the wider safety case for nuclear instrumentation and control systems, could provide further evidence concerning confidence in the safe design and deployment of these systems, e.g., enhanced confidence in the software testing process for certain applications.

C. Study Objectives

The nuclear industry partners were interested to understand if mutation testing might be beneficial in the development of safety-critical software (supporting the "production excellence" argument) or, alternatively, whether it would be useful in the independent assessment of a COTS software-based device (supporting the "confidence building" argument). Accordingly, the objectives of our study were as follows:

1. To determine whether the current standards in the nuclear industry for test assurance could be made more rigorous by the application mutation testing;
2. To determine whether cost reduction techniques proposed in the literature can make mutation testing affordable for the nuclear industry whilst retaining its power; and
3. To determine the most effective mutation operators for a typical nuclear software system.

D. Evaluation Criteria

The following criteria were important for the study to be valid:

(a) To meet objective (1), we needed to use mutation testing to assess an appropriate case study – a test suite satisfying current nuclear industry standards.

(b) To meet objective (1) we needed an appropriate comparison measure for our claims about relative fault-finding adequacy of different test coverage criteria.

(c) Additionally, in order to meet objective (2), we needed an appropriate measure of the additional costs incurred by using mutation coverage as our test suite adequacy criteria.

(d) To meet objective (3), we needed an appropriate measure to calculate the relative fault-finding ability of each mutation operator.

In this paper, we achieved criterion (a) by measuring the mutation coverage achieved by a test suite that satisfies branch coverage. This is appropriate as branch coverage is widely applied in the nuclear industry, being mandated for example by *IEC 61508* for Safety Integrity Level (SIL) 3 systems [5]. In *IEC 61508*, SILs specify safety requirements and measures that are allocated to safety functions in order to justify confidence (on a scale of 1 to 4) that the functions will not fail (particularly due to systematic causes for software components). The allocation of SILs depends on the necessary risk reduction in order to achieve tolerable risk, considering both the frequency of the hazardous events and their consequences.

We achieved criterion (b) by determining the mutation score achieved by the original branch-adequate test suite against our mutant set. The mutation score is widely used to assess test suite quality, and our use of TCE (in Section IV) meant that our mutation scores were very accurate.

We achieved criterion (c) by comparing the size of the test suites needed to achieve branch coverage and 100% mutation score, respectively. Here, we worked on the assumption that the cost of test development and maintenance is roughly proportional to the number of test cases developed. Using the mutation score alone is not enough, as one test case may kill more than one mutant (thus making the increase in test suite size less than the raw difference in mutation score would suggest).

We achieved criterion (d) by performing an analogous study to that used to achieve criterion (b) but determining the percentage of surviving mutants produced by each mutation operator.

E. Nuclear Software System

We performed a mutation testing process on a real nuclear software system. Specifically, this study simulates a complete mutation testing process applied to a COTS software-based device developed by a supplier to the UK nuclear industry. The device is used throughout the industry in a range of safety applications. The device receives a variety of inputs from field sensors and carries out user-configurable computations to deliver the required safety outputs. The overall COTS system (including the hardware and firmware) was developed in accordance with *IEC 61508* to satisfy the requirements of SIL 3 and achieves 100% branch coverage. MC/DC was not used during the development of the firmware, in line with the requirements of the standard. An interesting aspect of using such software is that, apart from satisfying a key coverage criterion, the system has undergone a thorough testing process that includes several forms of assurance. This fact differentiates this study from other previous evaluations related to mutation testing, which are mostly based on test suites developed without using the guidelines mandated by a standard. The firmware is implemented in the C programming language, following MISRA C coding standards [33], which are widely used in such systems as they are devoted to improve the safety and reliability of the code, We compiled the code with *gcc 5.4.0* in a machine running *Ubuntu 16.04*.

We selected 15 functions from different modules of the firmware. We should note that, overall, there are many modules with functions of similar structures. As such, to avoid biased results derived from the injection of mutations into similar code structures, we carefully studied the code to include functions with different functionalities, use of language facilities, size, and cyclomatic complexity [36] It is important to note that most of the functions in the firmware were composed of very few lines because the code had been implemented with utmost modularity and was highly optimised. As a result, the size of the selected functions ranged from 10 to 63 lines of code and their cyclomatic complexities ranged from 2 to 9.

For the sake of confidentiality, we will refer to this program pseudonymously as *project*, and its functions as *F1*, *F2 ... F15*. Table I shows several characteristics of the functions selected for this study.

The project was supplied with a unit test suite for each of the analysed functions (input values and expected outcome). All the test cases in these suites passed successfully when executed.

TABLE I:
FEATURES OF THE FUNCTIONS UNDER STUDY IN THE PROJECT

Source	**Features**		
Function	***Lines of code****	***Cyclomatic complexity***	***Number of test cases***
F1	14	2	25
F2	34	6	15
F3	38	7	15
F4	10	2	96
F5	40	8	13
F6	63	7	20
F7	27	4	9
F8	38	4	8
F9	15	3	7
F10	54	9	25
F11	29	5	5
F12	42	7	5
F13	32	4	16
F14	32	4	32
F15	16	3	11

* *Lines of code counted with c_count as "lines containing code"*

F. Mutation Tool

It was important for this study to use a relevant mutation tool — a tool that is practical for real-world use, or at least representative of such tools.

At present, there are a variety of mutation tools for C with different features, according to the survey by Jia and Harman [3]. While several of them are commercial[1] or are not publicly available, five of them are accessible.

Among these tools, *Proteum/IM 2.0* [38] and *MILU* [39] are the most widely used in other research studies, such as to evaluate selective mutation [14] or higher order mutation [12]. While *MILU* offers fewer features than *Proteum*, it automates most of the mutation analysis process, in contrast to *Proteum* which requires considerable manual intervention [40].

For the purpose of this study, we selected *MILU 3.2*[2], the most recent version available online when we performed the experiments.

G. Cost Reduction Techniques

Several techniques have been proposed to reduce the expense of mutation testing. These techniques have been extensively studied in academia (see Section II.A.2), but less so in realistic industrial applications. There is consequently a lack of evidence of their applicability to industrial systems, or information concerning their effectiveness.

In this study, we apply two cost reduction techniques: operator-based Selective Mutation (SM) and Trivial Compiler Equivalence (TCE). We expect SM to reduce the number of mutants generated, and TCE to remove ineffective (invalid, equivalent, or duplicate) mutants. As SM is almost universally recommended, we used it in all our experiments; as TCE is a fairly new technique, we repeated our experiments with and without it.

1) *Selective Mutation*

In SM [13], only some of the operators are applied (while the rest are discarded) under the premise that this subset of operators is representative of the full set of mutants.

The survey by Delahaye and du Bousquet [40] states that *MILU* implements the 77 mutation operators that Agrawal et al. [17] identified for C. However, the number of mutation operators has been reduced in *MILU 3.2* — following studies on SM [13], [14], only 12 of those 77 operators were included in this version. Two additional operators are also included: *SSDL* (delete statements) and *SBRC (*replaces *break* by *continue)*. Table II presents the operators that were used in the study – 10 of the 12 from *MILU*'s "Selective" set and one of the available two from its "Other" set. Note that:

- We excluded the "arithmetic assignment operator" (OOAN) and "bitwise logical assignment operator" (OBBA) because they can be substituted in the code by plain arithmetic and bitwise logical operators respectively, maintaining the same functionality. By doing this transformation, OAAN and OBBN apply in those cases.
- We included SSDL because recent studies have pointed to the usefulness of this operator [41]. Moreover, the comparable study by Baker and Habli [32] analyses this operator, so it is interesting to observe if mutants from this operator are also effective in detecting deficiencies in our branch-adequate test suites.

[1] A brief description of these commercial products can be seen in the study by Delgado-Pérez et al. [37].

[2] http://www0.cs.ucl.ac.uk/staff/y.jia/Milu/, last accessed 17/06/2017

TABLE II:
LIST OF MUTATION OPERATORS APPLIED

Operator	**Description**	**Modifications**
From MILU's "Selective" set		
CRCR	*Integer* constants replacement	9 insertions (0, 1, -1 and 6 additional values)
ORRN	Relational operator replacement	5 replacements (==, !=, <=, >=, < and >)
OAAN	Arithmetic operator replacement	4 replacements (+, -, *, / and %)
OLLN	Logical operator replacement	1 replacement (&& and ||)
OLNG	Logical negation	3 replacements (*x op !y*, *!x op y* and *!(x op y)*, where
OCNG	Logical context negation	1 replacement (applied to *if* and *while* statements)
OIDO	Increment/Decrement replacement	3 replacements (increment/decrement and
OBBN	Bitwise logical replacement	1 replacement (& and |)
UOI	Unary operator insertion	4 insertions (increment/decrement and prefix/postfix)
ABS	*Integer* and *float* variable absolute value	2 insertions (abs() and -abs())
From MILU's "Other" set		
SSDL	Statement deletion	1 deletion

- We excluded SBRC because, unlike SSDL, there are no recent studies that suggest it is useful.

SM is easy to implement in any mutation tool: either the tool provides the tester with a reduced subset of mutation operators (as in *MILU*) or it adds the option to enable/disable mutation operators.

2) *Trivial Compiler Equivalence*

Recently, *MILU* has been improved by incorporating TCE, which we briefly described in Section II.A.2. TCE allows the detection of three classes of ineffective mutants (mutants that waste time and/or distort the achieved mutation scores):

- All *invalid* mutants — those that cannot be compiled.
- Some *equivalent* mutants — those that have the same external behaviour as the original function.
- Some *duplicate* mutants — those that have the same functionality as another mutant in the set.

TCE works by comparing the binary files produced by the *gcc* compiler. A mutant is noted as invalid if it does not compile. A mutant is marked as equivalent if there is no difference between the binary files originated from the original program and the mutant. Two mutants are considered to be duplicate when their binary files are the same. Using the same example as the one to explain equivalence, depending on the compiler and the level of optimisation, TCE may determine that a mutant with the code "*if (x > 1) x = 1;*" and a mutant with the code "*if (x >= 1) x = 1;*" are behaviourally equivalent and are, therefore, duplicate mutants.

It is possible to apply different levels of optimisation when compiling, which can lead to different binary files and thus different results from TCE. It is reasonable to expect that more aggressive optimisation will lead to greater effectiveness of TCE, as many optimisations work by eliminating code elements that do not affect the output. This is not, however, guaranteed. For detecting equivalent mutants, the experiments by Papadakis et al. [23] show no clear winner among *gcc*'s optimisation options. For detecting duplicate mutants, Papadakis et al. observe that the best options are –O2 and –O3.

In this study, we initially focused on *gcc*'s highest level of optimisation (–O3) to learn about the limits of the application of this technique. According to the *gcc* documentation, this is reliable for all standards-compliant C programs. However, due to odd behaviour noted during the experiments, we repeated TCE on one function without optimisation (see Section IV.B).

TCE can plausibly be applied in real-world contexts. There is some cost in compute time, which corresponds to:

- The compilation process when applying an optimisation setting (in the experiments by Papadakis et al. [23] this was almost five times higher for –O3 than using no optimisation option);
- The equivalence detection process (one comparison between the binary of each mutant and the original program); and
- The duplicate mutant detection (each mutant is compared with the other mutants generated in the same function).

The results reported by Papadakis et al. [23] with respect to efficiency suggest that TCE is reasonably fast even for the most aggressive optimisation option: there is a trade-off between the compilation time and the effectiveness of the optimisation option, and the time required for equivalent and duplicate mutant detection is quite small when compared to the compilation time.

III. EVALUATING BRANCH-COVERAGE-DRIVEN TESTING USING SELECTIVE MUTATION

A. *Experiment*

In this first experiment, we carried out the following steps for each of the functions listed in Table I:

1. We generated all mutants from our set of mutation operators (see Table II). This set naturally included invalid, equivalent and duplicate mutants.
2. We ran the exiting test suite against the mutants. This allowed us to classify the mutants as killed or alive.
3. We calculated the mutation score based on the findings from the previous step.

B. *Results*

Table III shows the number of operators applied and the distribution of mutants in each of the functions, and can be summarised as follows:

- 2,509 mutants were generated in total, F1 and F6 being the functions with the smallest and largest number of mutants respectively (75 and 331 mutants);
- 81.94% of those mutants were killed (2,056) and 453 mutants were alive, with every function having some live mutants;
- The mutation score ranges from 56.32% in F2 to 96.10% in F8. None of the functions achieved 100% mutation coverage.

However, we have to note that these are only preliminary results because we have not tackled some factors that might affect the mutation score (which will be addressed in the next section).

TABLE III:
INITIAL DISTRIBUTION OF MUTANTS IN THE FUNCTIONS UNDER STUDY

	Generation		**Test execution**		**Analysis**
Function	***Operators applied***	***Total Mutants***	***Killed***	***Alive***	***Mutation score***
F1	7	75	59	16	78.67
F2	6	277	156	121	56.32
F3	5	118	84	34	71.19
F4	7	76	61	15	80.26
F5	7	164	113	51	68.90
F6	8	331	310	21	93.66
F7	5	158	148	10	93.67
F8	5	154	148	6	96.10
F9	7	134	105	29	78.36
F10	9	200	170	30	85.00
F11	7	210	186	24	88.57
F12	6	212	188	24	88.68
F13	5	94	86	8	91.49
F14	7	186	141	45	75.81
F15	7	120	101	19	84.17
Mean	-	-	-	-	81.94
Total	-	2,509	2,056	453	-

C. *Implications*

From the above, it first appears that testing guided by branch coverage has led to relatively weak test suites (ones that may fail to identify many faults).

There are reasons to be suspicious of these results, however. There are three ways in which the mutation scores may be misleading:

- Some "killed" mutants may be invalid, i.e., not written in valid C code. The results above count invalid mutants as killed, but those mutants cannot actually be executed. These mutants may have increased the apparent mutation score as *MILU* does not differentiate invalid mutants from killed mutants. Test suites should not get credit for killing mutants that would be killed by the compiler anyway.
- Mutants that remained alive have not been inspected yet to determine whether there are some equivalent mutants. Therefore, there is the possibility that the final mutation scores are higher than stated — test suites should not be penalised for failing to kill equivalent mutants.
- There is also the possibility that some mutants represent the same fault (that is, they are duplicate). Depending on the nature of the fault (whether the test suite kills it or not), this could increase or decrease the reported mutation score.

IV. Evaluating Branch-Coverage-Driven Testing Using Selective Mutation and Trivial Compiler Equivalence

A. *Experiment*

For a more accurate mutation score, we carried out the following steps:

1. We applied TCE to detect invalid, duplicate and equivalent mutants automatically.
2. We ran the exiting test suite against the mutants not flagged by TCE. This allowed us to classify the mutants as killed or alive.
3. We manually inspected the remaining live mutants to identify equivalent mutants not revealed by TCE. To do this, we designed further test cases with the specific aim of killing all surviving mutants; those mutants that still remained alive after this process were finally determined as equivalent.
4. We calculated the mutation score based on the findings from the previous steps.

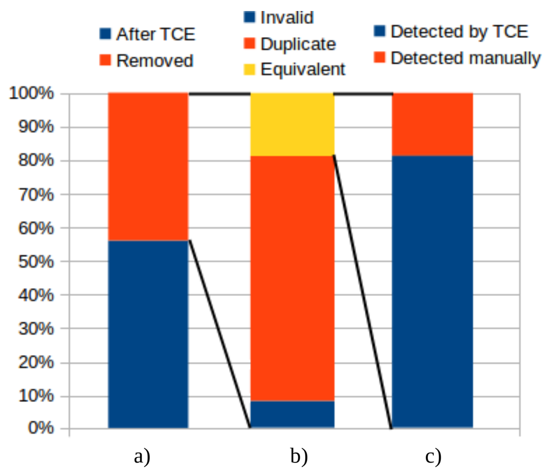

Fig. 1 - a) Proportion of mutants not detected as invalid, duplicate or equivalent by TCE ("After TCE") and detected ("Removed"); b) proportion of mutants in "Removed" that are "Invalid", "Duplicate" and "Equivalent" c) Proportion of "Equivalent" mutants "Detected by TCE" and "Detected manually" (manual inspection).

TABLE IV:
DISTRIBUTION OF MUTANTS AND MUTATION SCORE IN THE FUNCTIONS UNDER STUDY AFTER APPLYING TCE

	Generation		**Test execution**		**Analysis**		**Mutation sufficiency**	
Function	***Total Mutants***	***Mutants after TCE***	***Killed***	***Alive***	***Equivalent***	***Surviving***	***Non-equivalent***	***Mutation score***
F1	75	45	38	7	1	6	44	86.36
F2	277	114	70	44	4	40	110	63.64
F3	118	61	51	10	7	3	54	94.44
F4	76	51	44	7	7	0	44	100
F5	164	105	80	25	4	21	101	79.21
F6	331	207	203	4	3	1	204	99.51
F7	158	126	121	5	5	0	121	100
F8	154	112	110	2	1	1	111	99.10
F9	134	71	58	13	6	7	65	89.23
F10	200	110	103	7	3	4	107	96.26
F11	210	93	89	4	4	0	89	100
F12	212	123	119	4	2	2	121	98.35
F13	94	54	51	3	1	2	53	96.23
F14	186	89	70	19	0	19	89	78.65
F15	120	46	46	0	0	0	46	100
Mean	-	-	-	-	-	-	-	92.20
Total	2,509	1,407	1,253	154	48	106	1,359	-

B. *Results*

Fig. 1 bar (a) depicts the percentage of the total of mutants that were discarded using TCE (43.92%). Bar (b) drills into the mutants removed by TCE, showing that the highest reduction was obtained through duplicate mutants detection (73.14%), followed by equivalent (18.87%) and invalid mutants (7.99%). We are confident from the method that TCE removed all invalid mutants and most duplicate mutants. We are confident from our inspection of surviving mutants that TCE removed 81.25% of all equivalent mutants. This is shown in bar (c) of Fig. 1.

Table IV furnishes a complete breakdown of the classification of mutants in each function and is categorised as follows:

- ***Generation***:
  - Total mutants: number of mutants generated initially.
  - Mutants after TCE: number of mutants once TCE has been applied to detect invalid, duplicate and equivalent mutants automatically.
- ***Test execution:*** *(from "Mutants after TCE")****:***
  - Killed: number of mutants detected by the test suite.
  - Alive: number of mutants not detected by the test suite.
- ***Analysis:*** *(from mutants in "Alive")*:
  - Equivalent: number of mutants manually identified as equivalent.
  - Surviving: number of mutants that the test suite fails to detect.
- ***Mutation sufficiency:***
  - Non-equivalent: number of mutants identified to represent a valid fault in the program; calculated as *"Mutants after TCE"*-*"Equivalent"*.
  - Mutation score: calculated as: *("Non-equivalent"-"Surviving")/"Non-equivalent" x 100*.

We can observe from this table that:

- In four functions (F4, F7, F11 and F15) the branch-adequate test suite is mutant-adequate (i.e., the mutation score is 100%);
- In the remaining 11 functions, the mutation score varies from 63.64% (F2) in the worst case to 99.51% (F6) in the best case;
- The percentage of live mutants (154) that turned out to be equivalent (48) is 31.17%. That means that around 70% of the live mutants can guide us on the creation of new test cases (106 mutants).

There was one strange false-positive — for one function (F7), TCE classified six mutants as equivalent even though they were killed by the original test suite (and thus cannot have actually been equivalent). Those six mutations were all generated by *CRCR* in the same location. In response, we removed the optimisation flag (–O3) when compiling this function; thereafter, those mutants were not marked as equivalent. The "F7" row in the above table uses the results of this unoptimised compilation (the classification of the rest of the mutants in that function remained the same when the flag was removed).

Papadakis et al. [23] do note that compiler settings may influence equivalence detection, but they suggest that this should only cause false negatives. They claim that their technique (correctly implemented) cannot classify a non-equivalent mutant as equivalent. Therefore, they appear to stem from faults in the TCE implementation or the tools it depends upon. This issue would merit further investigation to establish its cause and how it can be avoided.

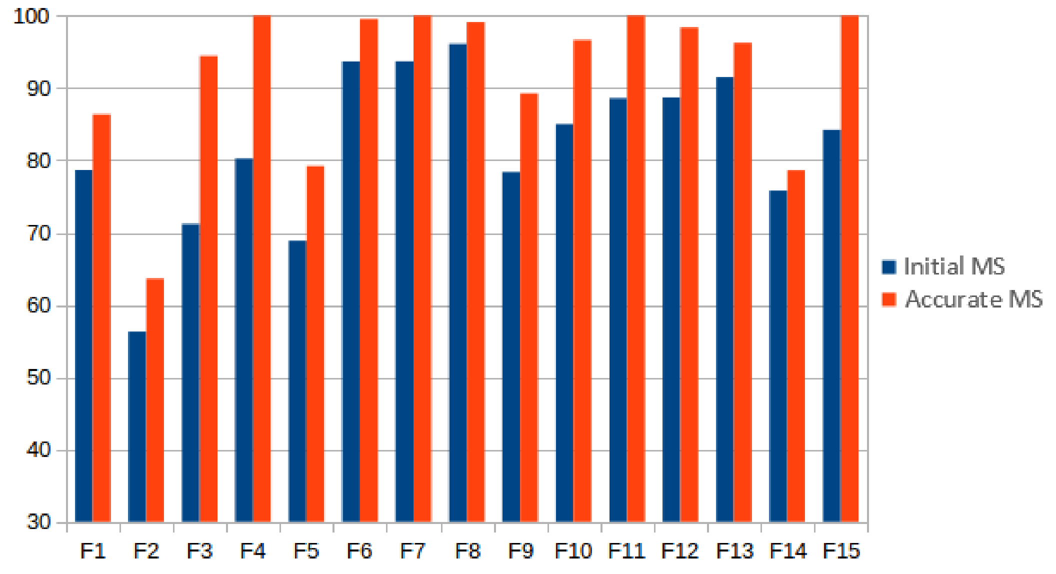

Fig. 2 - Initial vs accurate mutation score in each of the functions under study

C. *Implications*

TCE has the potential to reduce the cost of mutation testing considerably. From the 2,509 mutants initially generated with nine mutation operators in 15 functions, only 1,407 mutants (56.08%) represent valid, unique and non-equivalent mutations in terms of automatic detection. From that reduced set, only 154 mutants had to be reviewed to determine whether they were equivalent or not (see column “Alive” in Table IV) far from the 453 mutants that remained alive without applying TCE (see column “Alive” in Table III). This is the most time-consuming and laborious task in mutation testing, and the costs of this step have been reduced here by about two-thirds.

Of the 154 surviving mutants, only 48 mutants were later manually identified to be equivalent. 81% of the equivalent mutants were directly detected by TCE.

We can now be fairly confident of the accuracy of our kill scores. Overall, the kill score was 92.20%; 106 out of 1,359 non-equivalent mutants (7.80%) were not killed by the current test suite. Contrast this with the pre-TCE kill score of 81.94%. Without TCE, our estimate of surviving mutants was more than twice its correct value.

The difference in accuracy between our pre- and post-TCE results is even more pronounced for a subset of the functions. Fig. 2 graphically depicts the initial and accurate mutation scores in each of the functions. The highest difference is found in F3, with a gap of 23.25 percentage points between the two scores.

In the experiments conducted by Papadakis et al. [23], TCE detected from 9% to 100% of all the equivalent mutants when considering the results in the subject programs individually. This means that the performance of TCE greatly varies depending on the features of the system under test. The proportion of equivalent mutants detected by TCE in this case study (81.25%) is within the range identified by Papadakis et al., but is far above their average (30%).

The mutation score provides an estimation of the weaknesses detected in a test suite. Over 73% of the functions present surviving mutants. While in F6 and F8 the mutation score is close to 100%, other functions have lower scores. F2, F5 and F14 are especially low: 63.64%, 79.21% and 78.65% respectively. This suggests that if we have a test suite that complies only with branch coverage, we may be unable to detect a large variety of faults in our code.

V. Evaluation of Mutation-Driven Testing

A. *Experiment*

Surviving mutants represent potential deficiencies in the test suite. Therefore, the more surviving mutants, the lower the suite’s ability to detect faults in the software. Surviving mutants can be used to design new test cases, thereby increasing the fault-revealing power of the suite. In this phase of the work, we inspected the uncovered surviving mutants and manually generated new test cases until all surviving mutants were killed (i.e., until we had a mutant-adequate test suite). In all cases, we achieved this by copying existing test cases and changing some input values — we did not need to create new test case structures.

As noted in Section II.D, our metric for cost of improvement was the increase in the number of test cases needed. There are, however, problems with this measure. A test case tailored to a specific surviving mutant may also kill other mutants at the same time. Similarly, a test case from the original suite may kill two or more of the killed mutants. This makes it difficult to calculate how much the test suite needed to be enlarged, or how large it would have been had mutation testing been used to create it.

TABLE V:
TEST SUITE IMPROVEMENT OF THE FUNCTIONS UNDER STUDY

Function	Original	MM branch-adequate	New test cases generated	MM mutant-adequate	New test cases in MM mutant-adequate	% of mutation-adequate test cases that are new
F1	25	4	5	7	4	57
F2	15	5	13	15	11	73
F3	15	11	3	11	3	27
F4	96	5	0	5	0	0
F5	13	7	8	11	5	45
F6	20	5	1	6	1	16
F7	9	5	0	5	0	0
F8	8	4	1	5	1	20
F9	7	5	5	7	2	28
F10	25	9	4	10	3	30
F11	5	4	0	4	0	0
F12	5	5	2	5	2	40
F13	16	4	2	4	2	50
F14	32	5	10	10	6	60
F15	11	6	0	6	0	0
Mean	20.1	5.6	3.6	7.4	2.67	36
Total	302	84	54	111	40	-

We compensated for the above problem by minimising both the original and improved test suites. A test suite is *minimal* when there are no smaller test suites killing all non-equivalent mutants. The minimisation of the test suite removes redundant test cases, leaving both the original and improved suites as accurate representations of what is needed. To perform this minimisation, we employed the algorithm used by Estero-Botaro et al. [42], which produces test suites that are exactly minimal rather than approximate. Therefore, in our experiment we minimised the test suites preserving mutation adequacy, obtaining *mutation-minimal test suites* (abbreviated as MM test suites from now on).

B. *Results*

The test-suite refinement process, in principle, was not as laborious as expected. In our experience, once we had identified the reason why a given mutant was not killed by the test suite, it was straightforward to construct a suitable test to kill it. In addition, when two or more functions followed a similar design pattern, we were often able to use the same reasoning to kill a mutant in all of them.

Determining equivalence with high confidence was difficult in some cases but, following the application of TCE, the number of potentially equivalent mutants was modest (48 in total). We expect that both identifying equivalence and following a mutation-driven testing process would be even less difficult for someone who is completely acquainted with the system under test.

Table V shows the results of improving the test suite to achieve mutation adequacy. The columns are as follows:

- **Original** — the size of the original branch-adequate test suite we were given by the developer.
- **MM branch-adequate** — the size of a MM version of the branch-adequate test suite, i.e., the minimal number of test cases needed to kill the same mutants as the whole branch-adequate test suite.
- **New test cases generated** — the number of new test cases generated to improve the MM branch-adequate suite into a MM mutation-adequate suite. Note that the number of additional test cases does not match the number of surviving mutants because a single additional test case sometimes kills a group of mutants.
- **MM mutant-adequate** — the size of a MM test suite that is also mutation adequate (i.e., kills all non-equivalent mutants).
- **New test cases in MM mutant-adequate** — the number of new test cases appearing in the minimal version of the mutation-adequate test suite.
- **% of mutation-adequate test cases that are new** — percentage of new test cases appearing in the minimal mutant-adequate test suite.

Note that in some cases the minimal size is the same for both branch-adequate and mutant-adequate test suites, even when new test cases are used in the mutation-adequate test suite (specifically F3, F12 and F13). This occurs when some of the new test cases can kill some of the mutants killed by the branch-adequate test suite. In this case, some test cases in the branch-adequate test suite are replaced by new test cases in the mutant-adequate test suite to maintain minimality. This is reflected in the column *"New test cases in MM mutant-adequate"*, which states how many cases in the mutant-adequate suite are genuinely new.

As can be seen from this data, the test suite has been augmented in the 11 functions with surviving mutants, especially in F2 and F14, where the MM mutant-adequate test suite contains 73% and 60% of new test cases respectively. On average the MM mutation-adequate test suites contained 36%

new test cases, with the remainder being ones from the originally supplied test suites.

In terms of additional test-development effort, the MM branch-adequate test suites had a total of 84 test cases; 54 new test cases had to be generated to achieve mutation adequacy. Assuming an equal effort to develop each test case, that is an extra 64% effort. However, had mutation adequacy been a target from the start, only 111 test cases would have been required (an increase of only 27 test cases from the MM branch-adequate test suite). This corresponds to an increase of 32% in testing effort.

However, branch coverage was not the only goal in generating the original test suites. They will have taken account of system requirements, other process compliance requirements, and developer ideas about likely faults. Therefore, a better estimate of additional effort to achieve mutation coverage would be the increase in the number of tests needed versus the original test suite count. This was calculated as (302+54 = 356) divided by 302 original tests, so 18% extra effort.

C. *Implications*

Once we had identified the surviving mutants, it was straightforward to create test cases that killed them. This led to a net increase in test case count of 18% versus the original branch-adequate suites. Given the potential fault-finding benefits compared to this extra effort (recall that the increase in the test suite size allowed us to kill 106 surviving mutants in the analysed functions), it is certainly plausible that mutation-driven testing will be worthwhile for safety-critical software.

VI. Evaluation of the Contribution of the Different Mutation Operators

A. *Experiment*

In this study, we used a reduced set of mutation operators by applying selective mutation. Despite this fact, it is interesting to know which of those operators are the most effective as the cost of applying mutation testing is, in part, dependent on the number of operators used.

In this experiment, we carried out the same steps as in Section IV.A, but we computed the results per mutation operator instead of per function analysed of the project. We should notice that, instead of the mutation score, in this case we show the "survival rate", which is computed as "100 – mutation score". This measure helps interpret the results: the higher the survival rate, the more effective the mutation operator.

B. *Results*

Table VI presents the distribution of mutants for each mutation operator, much as in Table IV. In this table, we also show:

- **Functions applied**: number of functions in which these operators generated at least one mutant.
- **Functions with surviving mutants**: number of functions in which these operators generated at least one surviving mutant.

All the mutants were generated by 9 operators of the selective set. CRCR was by far the operator generating the highest number of mutants and it was applied to all the functions (as were SSDL, ORRN and OCNG). On the other hand, OBBN was the less prolific (17 mutants) as well as the least-applied operator in the set (3 functions). CRCR, ORRN and OANN (the most prolific operators) were the operators that generated the 48 equivalent mutants not detected by TCE.

The results show that five out of nine operators (SSDL, CRCR, ORRN, OAAN and OLLN) generated mutants that survived the execution of the test suite. The results also suggest that the operators OLNG, OCNG, OIDO and OBBN were not effective operators for the system under test.

Judging by the survival rate, OLLN seems the most valuable operator. However, we should also consider the following factors:

- **Mean** — computed as the sum of the survival rates of the operator in each function divided by "Functions applied";
- **Standard deviation** — standard deviation in the survival rates in the functions in which the operator was applied;
- **% Functions with surviving mutants** — calculated as: "Functions with surviving mutants" / "Functions applied".

This information can be graphically seen in Fig. 3. The mean shows that ORRN (9.1), CRCR (8.9) and OAAN (7.2) produce more surviving mutants in the functions on average than OLLN (6.7). Similarly, the standard deviation of OLLN (21.1) indicates that the number of surviving mutants from this operator is substantially dissimilar in these functions (in fact, only one of the 10 functions in which OLLN is applied presents surviving mutants). ORRN (10.1) followed by OAAN (12.6) are the most stable operators according to the standard deviation. Finally, the percentage of functions with surviving mutants confirms ORRN as an effective operator, since some mutants produced by this operator survive in 60% of the functions in which this operator is applied. CRCR and OAAN tie in the second position (40%), while OLLN (10%) is at the bottom of this classification.

C. *Implications*

From the initial set of mutation operators, only five operators (SSDL, CRCR, ORRN, OAAN and OLLN) generated some mutants not detected by the branch-adequate test suite. This subset of operators aligns with the results reported by Baker and Habli [32], where these same five operators generated surviving mutants when analysed with test suites achieving statement-level coverage in a project implemented in C. In that study, the operators CRCR, ORRN and OAAN also generated some mutants that survived the execution of a test suite achieving MC/DC coverage in a project implemented in Ada. That fact suggested that the operators SSDL and OLLN were not fruitful with more demanding coverage levels. However, the results in this paper have shown that these two operators are still useful when it comes to branch coverage.

TABLE VI:
DISTRIBUTION OF MUTANTS AND SURVIVAL RATE OF EACH MUTATION OPERATOR

	Generation			**Analysis**			**Mutation sufficiency**	
Operator	***Total Mutants***	***Functions applied***	***Mutants after TCE***	***Equivalent***	***Surviving***	***Functions with surviving mutants***	***Non-equivalent***	***Survival rate***
SSDL	211	15	127	0	8	4	127	6.3
CRCR	1,386	15	649	18	51	6	631	8.1
ORRN	425	15	292	27	32	9	265	12.1
OAAN	296	10	224	3	11	4	221	5.0
OLLN	25	10	19	0	4	1	19	21.1
OLNG	75	10	22	0	0	0	22	0
OCNG	56	15	53	0	0	0	53	0
OIDO	18	5	6	0	0	0	6	0
OBBN	17	3	15	0	0	0	15	0

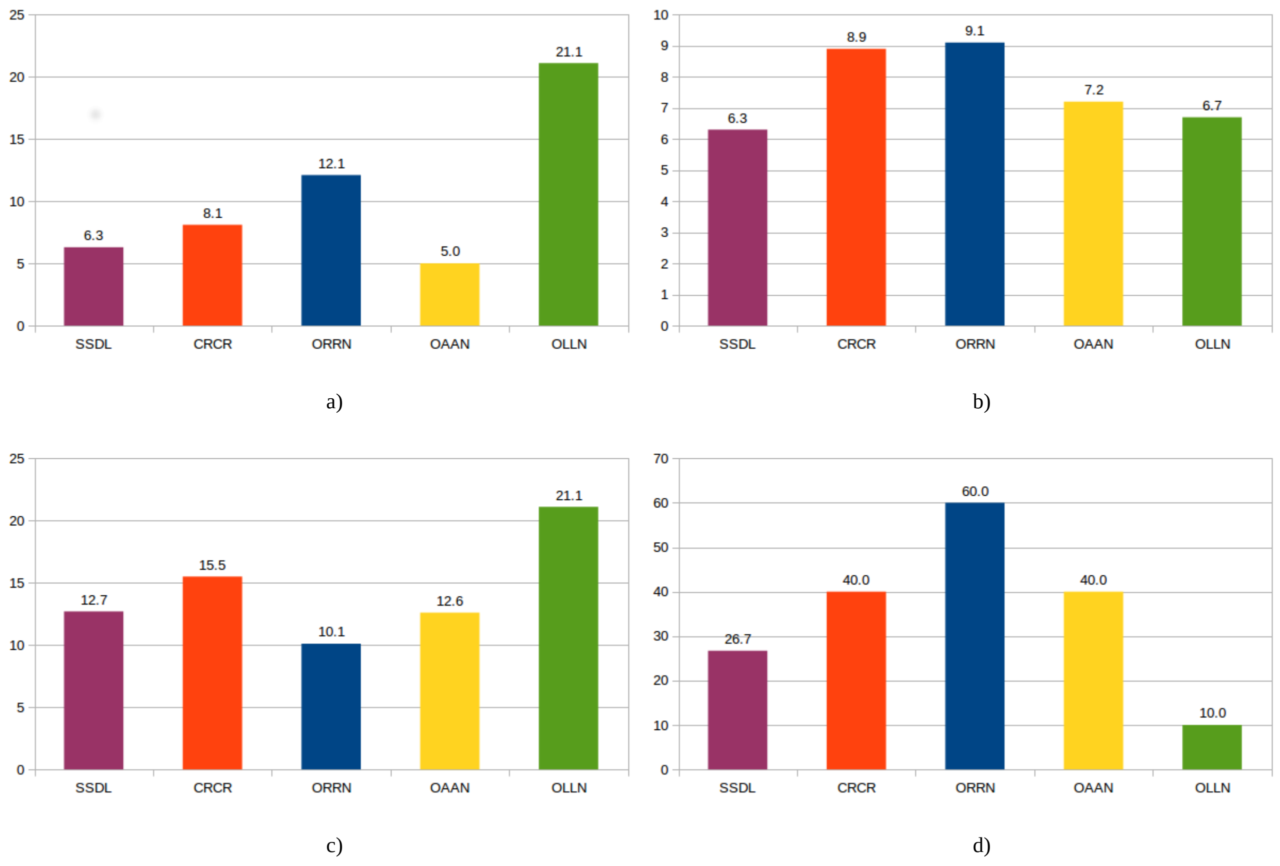

Fig. 3 – Mutant survival statistics of each mutation operator that generate surviving mutants:
a) raw survival rate; b) mean survival rate; c) standard deviation of survival rate; d) % functions with surviving mutants.

On the basis of the results, ORRN seems the most valuable operator, as it was in the study by Baker and Habli [32]. We have also shown how important is to measure the relative value of each operator in terms of the mean and the standard deviation (of the survival rates in the functions) and the percentage of functions with surviving mutants.

VII. DISCUSSION

A. *Overall assessment given industrial requirements*

Our results indicate that mutation testing is both effective and feasible for a safety-critical system. The experiments highlighted potential weaknesses in the test process despite meeting a key test coverage criterion that is specified in a primary safety standard, *IEC 61508*, and despite meeting the stringent expectations of the nuclear industry. Mutation testing is also shown to be practical in the sense that we utilised already available techniques (SM and TCE), and a publicly available mutation analysis tool (*MILU*).

Of course, the system considered in this research covers one case only, i.e., one software system in one industrial context. However, this system is representative of many applications in the nuclear industry in the sense that it is developed in a high-integrity language subset, i.e., MISRA C, and against the requirements of *IEC 61508*. Both MISRA C and *IEC 61508* are highly regarded and widely used in the nuclear industry and within the safety-critical domain generally. For example, the requirements within *IEC 61508* have formed the basis for safety standards in different domains including automotive [5] and railway [6], particularly with regard to the test coverage criteria (statement coverage, branch coverage and MC/DC).

Ideally, we would also measure the mutation coverage achieved by the more thorough MC/DC structural criteria. However, we did not have access to such test suites. If we wanted to replicate the study with test suites that met MC/DC, we would need to develop new artificial test suites. That would break the research design methodology followed in this study since the data used is based on real-world test suites. Therefore, the results of a study with MC/DC could not be fairly compared with the rest of the results shown in the paper.

The primary issue might not lie in whether mutation testing is effective or feasible. Rather, it is in the extent to which it is cost-effective, given the potential increase in confidence relative to other competing techniques, e.g., formal methods or runtime verification. Another issue lies in whether mutation testing is effective when the results of the mutation testing process, which is based on hypothetical faults, are compared to actual faults reported for safety-critical operations. Progress towards any wider industrial adoption will rely on a more explicit consideration of these issues. As such, the primary contribution of our paper is in showing the *potential* effectiveness and feasibility of the technique based on real world code and data.

B. *Feedback from industry*

The feedback from the nuclear industry partners has been positive. The engineers were surprised that the branch-adequate test suite was shown to be comparatively weak in its fault detection capability, although it was recognised that this was not necessarily a cause for concern as no coding errors were identified during the experiments and the overall device had been subjected to other forms of assurance including statistical testing. They were also interested in the modest level of additional effort to move from a branch-adequate test suite to a mutation-adequate test suite. It was interesting that, after studying in detail surviving mutants, repeatable test deficiencies in the original test suites could be identified. For instance, following the application of ORRN and CRCR, we found out that the boundary conditions were not fully tested in all functions.

Moving forward, the industry is keen to explore the potential power of a mutation-adequate test suite in comparison to a branch-adequate test suite. As such, future investigations are likely to focus on the application of a mutation-adequate test suite to a system seeded with real faults that passed through the original testing process. These investigations may seek to support or challenge the underlying theory of mutation testing.

C. *Threats to Validity*

Here, we present some caveats that slightly temper our claims, and some rebuttals of concerns readers might have regarding the validity of the study.

Construct validity: Our main measures of concern are testing power achieved and the associated cost. We derive metrics for these as the mutation score and the required number of test cases, respectively. Any weaknesses in these metrics endanger our construct validity.

The mutation score is a common metric in software testing research for determining fault-detection power. It can, however, be distorted by inclusion of duplicate, equivalent and invalid mutants. In this experiment, we removed these using automated and manual methods.

To generate our mutation score, we intended to use a reduced set of eleven mutation operators. This is the set of operators included in the mutation tool employed (*MILU*), based on studies about selective mutation for the reduction of mutants [13], [14]. However, only nine of these operators could be applied to the software under test, as the features of this software prevented two operators related to built-in types from generating some mutants (UOI and ABS). Had we implemented custom versions of those operators that worked with this codebase, it is possible that the mutation scores would have been slightly different.

The number of test cases is a debatable measure of test development cost. It is likely that there is a moderate correlation in practice, but also a high degree of error (for example, it is likely that earlier tests cost more than later tests because later tests can reuse code and techniques from earlier ones, as we found when expanding the test suite in this study). However, we are not aware of a practical measure that is a better proxy for cost.

When we look at the increase in number of test cases to move from branch to mutation coverage, there is a risk of ending up with redundant test cases. This can happen when a new test case introduced to kill a mutant also contributes to branch coverage in a way that completely subsumes one of the original test cases. To counter this threat, we minimised both test suites. However, recognising that any real-world test suite could (and should) be developed to cover diverse criteria (not just branch coverage or mutation adequacy), we have also calculated the proportional increase of the mutation-adequate test suite against the full original test suite. Thus, given that test cases were extended manually (differently from the method used in practice to generate test cases), the calculations shown in this paper should be treated as estimations.

Internal validity: In this paper, we have studied whether TCE reduces the number of invalid, duplicate and equivalent mutants, and how many extra tests are required to move from branch coverage to mutation adequacy. We are confident that TCE correctly removed all the invalid and most duplicate mutants. We are fairly confident that all the equivalent mutants it removed were indeed equivalent. However, as noted in Section IV.B, there was a case in which TCE classified some killable mutants as equivalent. Were the faults that caused this identified and fixed, the mutation score might be slightly different.

External validity: We have only used one target system, but it is representative of a range of real-world systems which comply with *IEC 61508* and MISRA C coding standards. Since these standards are applied for many safety-critical software systems, our results may apply to many such systems.

TCE is implemented as a combination of *gcc* (a collection of compilers for several programming languages) and the *diff* file comparison utility. As such, it may not be available on all platforms or for all languages. However, its implementation is straightforward to duplicate given a compiler and a file comparison utility that supports binary files.

VIII. Conclusion

This study presents an empirical evaluation of mutation testing in a joint project sponsored by the UK nuclear industry. Our primary conclusion is that mutation testing can assist in designing a better test suite for safety-critical software when compared to one that is designed to demonstrate compliance with a branch coverage criterion.

The results consistently show that a branch-adequate test suite fails to detect injected faults in the code. This happened in 11 out of the 15 functions analysed. In the most noteworthy case, 37% of the faults injected into one of the functions were not detected by the original test suite, and several new test cases could be added to improve its fault-detection capability.

The approach followed in this study (use of a selective set of mutation operators followed by the application of TCE to detect ineffective mutants) greatly reduces the cost of applying mutation testing. The mutants generated in the analysed functions ranged from 75 to 331, far fewer than those found in similar previous studies. Around 44% of these mutants were automatically discarded on average, and around 96% of the remaining mutants represented valid faults. By applying TCE, we were able not only to reduce the cost of identifying equivalent mutants (81% of the set of equivalent mutants) but also to measure the mutation score more accurately. The study about the surviving mutants generated by each mutation operator revealed that the set of operators may be reduced further without losing significant effectiveness in these systems.

As a result, the test suite improvement derived from mutation testing only requires a test suite increase of 18% compared with the original branch-adequate test suites. It is thus plausible that it will be worth its cost in many situations, especially in safety-critical systems.

To justify increased costs, however, industrial developers are likely to need more confidence that achieving mutation coverage will find more faults. As such, our industrial partners recommended conducting new experiments challenging mutation testing to detect real coding errors, specifically those that have not been found by existing tests. Given evidence of that, it is likely that many developers may be willing to accept the extra costs and adopt mutation testing as another approach to assuring safety-critical software.

Acknowledgment

This study was funded by the UK Control and Instrumentation Nuclear Industry Forum (CINIF). This paper was also partially funded by the European Commission (FEDER) and the Spanish Ministry of Economy and Competitiveness (National Program for Research, Development and Innovation) through the project DArDOS (TIN2015-65845-C3-3-R). We also thank Francisco Palomo-Lozano for helping us use his algorithm to find minimal test suites.

References

[1] R. G. Hamlet, “Testing programs with the aid of a compiler,” *IEEE T. Software Eng.*, vol. 3, no. 4, pp. 279–290, 1977.

[2] R. DeMillo, R. Lipton, F. Sayward, “Hints on test data selection: help for the practicing programmer,” *Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[3] Y. Jia, M. Harman, “An analysis and survey of the development of mutation testing,” *IEEE T. Software Eng*., vol. 37, no. 5, pp. 649–678, 2011.

[4] M. Usaola, P. Mateo, “Mutation testing cost reduction techniques: A survey,” *IEEE Softw*., vol. 27, no. 3, pp. 80-86, 2010.

[5] Functional safety of electrical/electronic/programmable electronic safety-related systems, IEC 61508, International Electrotechnical Commission, 2010.

[6] Road Vehicles: Functional Safety, ISO 26262, International Organization for Standarization, 2011.

[7] Software Considerations in Airborne Systems and Equipment Certification, DO-178C, RTCA, 2011.

[8] J. J. Chilenski, S. P. Miller, “Applicability of Modified Condition/Decision Coverage to Software Testing,” *Software Eng. J.*, vol. 9, no. 5, pp. 193-200, 1994.

[9] M. R. Woodward, “Mutation testing - its origin and evolution,” *Inform. Software Tech*., vol. 35, no. 3, pp. 163–169, 1993.

[10] A. J. Offutt, “Investigations of the Software Testing Coupling Effect,” *ACM T. Softw. Eng. Meth.*, vol. 1, no. 1, pp. 5–20, 1992.

[11] L. Zhang, S.S. Hou, J. J. Hu, T. Xie, H. Mei, “Is operator-based mutant selection superior to random mutant selection?,” in *Proc. ICSE*, Cape Town, South Africa, 2010, pp. 435–444.

[12] Y. Jia, M. Harman, “Higher order mutation testing,” *Inform. Software Tech.*, vol. 51, no. 10, pp. 1379–1393, 2009.

[13] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, C. Zapf, “An experimental determination of sufficient mutant operators,” *ACM T. Softw. Eng. Meth.*, vol. 5, no. 2, pp. 99–118, 1996.

[14] E. F. Barbosa, J. C. Maldonado, A. M. R. Vincenzi, “Toward the determination of sufficient mutant operators for C,” *Softw. Test. Verif. Rel*., vol. 11, no. 2, pp. 113–136, 2001.

[15] A. S. Namin, J. H. Andrews, D. J. Murdoch, “Sufficient mutation operators for measuring test effectiveness”, in *Proc. ICSE*, Leipzig, Germany, 2008, pp. 351–360.

[16] M. E. Delamaro, L. Deng, N. Li, V. Durelli, A. J. Offutt, “Growing a reduced set of mutation operators”, in *Proc. SBES*, Maceió, Brazil, 2014, pp 81–90.

[17] H. Agrawal, R. A. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, E. Spafford, “Design of mutant operators for the C programming language,” Software Engineering Research Center, Purdue University, West Lafayette, Indiana, USA, Tech. Rep. SERC-TR-41-P, 1989.

[18] K. Adamopoulos, M. Harman, R.M. Hierons, “How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution,” in *Proc. GECCO*, Seattle, WS, USA, 2004, pp. 1338–1349.

[19] R. M. Hierons, M. Harman, 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.

[20] A. J. Offutt, J. Pan. “Detecting equivalent mutants and the feasible path problem”, in *Proc. COMPASS*, Gaithersburg, MD, USA, 1996, pp. 224–236.

[21] M. Papadakis, M. E. Delamaro, Y. Le Traon, “Mitigating the effects of equivalent mutants with mutant classification strategies,” *Sci. Comput. Program*, vol. 95, no. P3, pp. 298–319, 2014.

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

[23] M. Papadakis, Y. Jia, M. Harman, Y. Le Traon. “Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique”, in *Proc. ICSE*, Florence, Italy, 2015, pp. 936–946.

[24] P. Ammann, M. E. Delamaro, A. J. Offutt, “Establishing theoretical minimal sets of mutants,” in *Proc. ICST*, Cleveland, Ohio, USA, 2014, pp. 21–30.

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

[26] X. Yao, M. Harman, Y. Jia, “A study of equivalent and stubborn mutation operators using human analysis of equivalence,” in *Proc. ICSE*, Hyderabad, India, 2014, pp. 919–930.

[27] L. Inozemtseva, R. Holmes, “Coverage is not strongly correlated with test suite effectiveness”, in *Proc. ICSE*, Hyderabad, India, 2014, pp. 435–445.

[28] M. Daran, P. Thévenod-Fosse, “Software error analysis: A real case study involving real faults and mutations,” in *Proc. ISSTA*, San Diego, California, USA, 1996, pp. 158–171.

[29] J. H. Andrews, L. C. Briand, Y. Labiche, “Is mutation an appropriate tool for testing experiments?,” in *Proc. ICSE*, St. Louis, MO, USA, 2005, pp. 402–411.

[30] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, G. Fraser, “Are mutants a valid substitute for real faults in software testing?,” in *Proc. FSE*, Hong Kong, China, 2014, pp. 654–665.

[31] R. Gopinath, C. Jensen, A. Groce. “Mutations: how close are they to real faults?,” in *Proc. ISSRE*, Naples, Italy, 2014, pp 189–200.

[32] R. Baker, I. Habli. “An empirical evaluation of mutation testing for improving the test quality of safety-critical software,” *IEEE T. Software Eng.*, vol. 39, no. 6, pp. 787–805, 2013.

[33] Guidelines for the use of the C language in critical systems, MISRA-C:2004, 2004.

[34] Safety Assessment Principles for Nuclear Facilities, Office for Nuclear Regulation, 2014.

[35] J. McDermid, T. Kelly, “Software in Safety Critical Systems-Achievement & Prediction,” *Nuclear Future*, vol. 2, no. 3, pp. 140, 2006.

[36] T. J. McCabe, “A complexity measure,” *IEEE T. Software Eng.*, vol. 2, no. 4, pp. 308-320, 1976.

[37] P. Delgado-Pérez, I. Medina-Bulo, J. J. Domínguez-Jiménez, “Analysis of the development process of a mutation testing tool for the C++ language”, in *Proc. ICCGI*, Seville, Spain, 2014, pp. 151–156.

[38] M. E. Delamaro, J. C. Maldonado, A. M. R. Vincenzi, “Proteum/IM 2.0: An integrated mutation testing environment,” in *Mutation testing for the new century*, Springer, Boston, MA, 2001, pp. 91–101.

[39] Y. Jia, M. Harman. “MILU: a customizable, runtime-optimized higher order mutation testing tool for the full C language”, in *Proc. TAIC PART*, Windsor, UK, 2008, pp. 94–98.

[40] M. Delahaye, L. Bousquet, “Selecting a software engineering tool: lessons learnt from mutation analysis,” *Software Pract. Exper.*, vol. 45, no. 7, pp. 875–891, 2015.

[41] L. Deng, A. J. Offutt, N. Li. “Empirical evaluation of the statement deletion mutation operator,” in *Proc. ICST*, Wellington, New Zealand, 2013, pp. 80–93.

[42] A. Estero-Botaro, F. Palomo-Lozano, I. Medina-Bulo, J. J. Domínguez-Jiménez, A. García-Domínguez, “Quality metrics for mutation testing with applications to WS-BPEL compositions,” *Softw. Test. Verif. Rel.*, vol. 25, no. 5–7, pp. 536–571, 2015.