Empirical Software Engineering manuscript No.
(will be inserted by the editor)

On the Use of Commit-Relevant Mutants

Miloš Ojdanić[1] · Wei Ma[1] · Thomas Laurent · Thierry Titcheu Chekam · Anthony Ventresque · Mike Papadakis

Received: date / Accepted: date

Abstract Applying mutation testing to test subtle program changes, such as program patches or other small-scale code modifications, requires using mutants that capture the delta of the altered behaviours. To address this issue, we introduce the concept of commit-relevant mutants, which are the mutants that interact with the behaviours of the system affected by a particular commit. Therefore, commit-aware mutation testing, is a test assessment metric tailored to a specific commit. By analysing 83 commits from 25 projects involving 2,253,610 mutants in both C and Java, we identify the commit-relevant mutants and explore their relationship with other categories of mutants. Our results show that commit-relevant mutants represent a small subset of all mutants, which differs from the other classes of mutants (subsuming and hard-to-kill), and that the commit-relevant mutation score is weakly correlated with the traditional mutation score (Kendall/Pearson 0.15-0.4). Moreover, commit-aware mutation analysis provides insights about the testing of a commit, which can be more efficient than the classical mutation analysis; in our experiments, by analysing the same number of mutants, commit-aware mutants have better fault-revelation potential (30% higher chances of revealing commit-introducing faults) than traditional mutants. We also illustrate a possible application of commit-aware mutation testing as a metric to evaluate test case prioritisation.

[1] co-first author, equal contribution and importance

Miloš Ojdanić · Wei Ma · Mike Papadakis
SnT, University of Luxembourg, Luxembourg
E-mail: {firstname.surname}@uni.lu

Thomas Laurent · Anthony Ventresque
Lero School of Computer Science, University College Dublin
E-mail: thomas.laurent@ucdconnect.ie, anthony.ventresque@ucd.ie

Thierry Titcheu Chekam
SnT, University of Luxembourg, Luxembourg
E-mail: thierry.titcheu.chekam@ses.com

Keywords mutation testing · change-relevant mutants · continuous integration · regression testing

Conflict of interest

The authors declare that they have no conflict of interest.

1 Introduction

Software systems are subject to regular modification during their life-cycle. Modifications are usually made in order to maintain and improve the software (fixing bugs, refactoring, or improving code quality), or to include new features. In either case, automated testing is used as gate-keeping, i.e., to establish confidence that the modifications did not break any of the previously developed program functionalities.

In such scenarios, developers often assume that the previous (operational) version of the system was stable and correct. Therefore, they are interested in testing only the behaviour delta of the changes they performed. This means that they want to assess the delta of behaviours between their pre- and post-commit system versions. For such cases developers need metrics quantifying the extent to which they have tested the error-prone program behaviours affected by their changes. Unfortunately, little research has been devoted to forming such change-aware test criteria. Change-aware test criteria would offer a viable, from an economic perspective, way of dealing with the continuous software modifications, as one would only focus on the particular program changes or commits.

Mutation testing has long been established as one of the strongest test criteria [50]. It operates by measuring the extent to which test suites can distinguish the behaviour of the original program from that of some slightly altered (syntactically altered) program versions, which are called mutants. Testers can use mutants to design strong test cases, likely to be fault revealing [2, 16] and to perform test assessment as it effectively quantifies the test suites' strengths [3].

Mutation testing research assumes a static nature of software, and thus it is focused on making the mutation score metric accurate with respect to all possible mutants that one can generate, by using a predefined set of mutation operators, in a given piece code. Thus, existing research is focusing on using specific mutant types [46]; on detecting equivalent mutants [30, 44], i.e., mutants that cannot be killed by any test case because they are semantically equivalent to the original program; or on eliminating redundant mutants [48, 34, 31], i.e., mutants that are killed "collaterally" whenever other mutants are killed [31] (subsumed by the subsuming mutants).

This strategy has the unfortunate effect of blindly using all possible mutants without considering their relevance to the task or to the most recent changes in question. To allow such focused testing, one should use only what we call commit-relevant mutants, i.e., mutants interacting with the changed program behaviours. These mutants are relevant to the program changes, meaning that they are killed by tests that exercise the committed code and its integration to the rest of the program under test. In terms of testing, these mutants form the change-relevant requirements and can be used to judge whether test suites are adequate in testing commits and, if not, to provide guidance in improving them (by creating tests that kill commit-relevant mutants).

In an attempt to form such commit-relevant mutants one could use the entire set of mutants or those that are located on the modified code, assuming that mutant locations reflect their utility and relevance. Unfortunately, such solutions are imprecise since they either include large volume of noise (irrelevant mutants), or are insufficient to cover all possible interactions between the unmodified and changed code. We argue that covering all interactions between unmodified and modified code is particularly important because problematic regression issues arise from such unforeseen interactions [8, 57]. This is demonstrated by our results, which show that the majority of the altered program behaviours is captured by mutants located on unmodified code parts. In fact the majority of the altered program behaviours are captured by mutants located on unmodified code parts.

This paper forms an extended study of our previous work [42], published at the 36th International Conference on Software Maintenance and Evolution (ICSME), which introduced and evaluated the concept of commit-aware mutation testing. Here, we extend the study by investigating the relationship of the commit-relevant mutants with other classes of mutants, i.e., subsuming and hard-to-kill mutants, and by demonstrating their use in controlled experiments. We thus, perform a use case that evaluates the ability of Regression Test Case Prioritisation techniques to reveal commit-relevant faults. The extended results demonstrate that commit-relevant mutants can offer useful insights in evaluating regression testing techniques.

Overall, the contribution of the paper regards the definition of the commit-relevant mutants and the related commit-relevant mutation-based test assessment. Intuitively, a mutant is commit relevant if it defines a test requirement (a mutant fault) that depends on the commit, i.e., the test cases that cover this requirement (detect this mutant fault) exercise the program behaviour altered by the commit. To ensure a testable link between the mutants and the commits, we require the existence of an "observable dependence" between the mutants and the committed code. This means that the presence and absence of the mutant and the commit code imply an observable behavioural change under some test execution.

We also show that by identifying those commit-relevant mutants one can accurately and adequately test program changes. Perhaps more importantly, we also demonstrate that mutation testing performed with the entire set of mutants or with the mutants located on the committed code is insufficient to assess how well subtle program changes have been tested. This implies that relevant mutants also enable the study of commit-aware fault detection assessment, in a sense using relevant mutants as a proxy for fault introducing commits. This aspect is missed by the software testing literature since it mainly focuses on using mutants as proxy of faulty program versions independently of the program changes under test. We showcase such a case by using commit-relevant mutants to evaluate regression test prioritisation techniques.

Taken together, the key research contributions of this present paper can be summarised as follows:

- We define commit-relevant mutation testing, which is based on the notion of commit-relevant mutants, i.e., mutants capturing the interactions between modified and unmodified code.
- We show that commit-relevant mutants are a distinct class of mutants, i.e., it differs significantly from the other mutant classes (subsuming and hard-to-kill mutants) [47].
- We investigate the extent to which mutation-based test assessment metrics such as a) the mutation score (score that includes the entire set of mutants), b) the delta of mutation scores between pre- and post-commit, c) the mutation score of mutants located on the committed code, correlate with the commit-relevant mutation score. Our results show that all three metrics have relatively weak correlations (less than 0.4), indicating the need for a commit-relevant test assessment metric.
- We further examine the potential guidance given by commit-relevant mutation testing by comparing the gains and losses of strategies that use the entire set of mutants, the mutants located on the committed code and the commit-relevant mutants. Our findings suggest that commit-relevant mutants have 30% higher fault revelation ability (w.r.t. real commit-introduced faults) than the other strategies when analysing the same number of mutants.
- We illustrate a possible application of commit-aware mutation testing as a metric to evaluate test case prioritisation.

2 Background

This section introduces background concepts, definitions, and work around mutation analysis, different ways of classifying mutants, and test selection.

2.1 Mutation Analysis

Test criteria are metrics quantifying the extent to which systems are tested [2]. They are based on the notion of test requirements, i.e., defining what should be tested. Depending on which test requirements are covered by a test suite, a test criterion defines a value that reflects how well it tests the system w.r.t. to the intended behaviour. Test criteria have been used to drive different aspects of the testing process, such as test generation [23] or test selection [64]. The test requirements are then used to decide which new tests are needed, or which tests are redundant. Test criteria can also be used to assess the thoroughness of a test suite, e.g. to decide if more effort should be devoted to testing or if sufficient confidence in the proper behaviour of the system has been gained. Test criteria are also used to assess other criteria [48].

Mutation analysis is a test criterion [39] that measures the capability of a test suite to detect artificial defects. Multiple versions of the program under test, called mutants, are created, that contain the artificial defects used as test requirements. The ability of the test suite to differentiate the program under test and these mutants is then measured. The artificial defects usually take the form of small syntactic changes in the code, such as changing “if $(a > b)$” into “if $(a \geq b)$”.

Mutants are systematically generated, following a set of replacement rules called mutation operators. Different mutation operators can be used in order to tailor the mutants created, and thus the test requirements. This allows the tester to focus on different aspects of the test suite. Similarly, these operators can be applied only to specific parts of the program, should the tester only want to focus on those.

Once mutants, i.e., test requirements, are created, the test suite is run against the program under test and the mutants in order to compare their behaviour. This behaviour is usually represented by the output of the program, captured by test or program assertions. If a test triggers different behaviours between the original program and a mutant, the mutant is considered to be “killed” (the test requirement represented by this mutant is fulfilled). A test killing a mutant not only shows that the test executed the mutant, but also that this execution resulted in an altered state, and that this alteration was propagated to the output of the program. If the original program and a mutant behave the same for all tests considered, the mutant is said to be “live”. The thoroughness of a test suite is measured using the “Mutation Score” (MS), the ratio of mutants killed by test suites over all killable mutants created.

2.1.1 Different Categories of Mutants: Killing all mutants is not feasible, as some mutants are semantically equivalent to the original program, i.e., will behave the same way for all possible inputs, although they are syntactically different. These mutants are called equivalent, while mutants for which there exists an input for which their behaviour is different from the original program's, are said to be killable.

When using mutation analysis to measure the thoroughness of a test suite, we do not want to take equivalent mutants into consideration, as even a perfect test suite will not kill them. Equivalent mutants have proven to be a major challenge in the area of mutation testing [50], as identifying them is an undecidable problem [9].

Even among killable mutants, not all mutants have the same value as test requirements. Some mutants constitute very easy requirements, that can be captured by almost any test, while some require specific tests to be crafted to kill them. This difference is captured by the notion of hard to kill mutants, based on the number of tests killing the mutant among the tests that reach it.

A mutant is hard to kill if it is killed only by a small proportion of the tests that reach it. On the other hand, if most tests reaching the mutant also kill it, it is easy to kill.

Interestingly, many killable mutants are equivalent to others, introducing skew in the Mutation Score. The studies of Papadakis *et al.* [49] and Kintis *et al.* [30] have shown this to be problematic and suggest getting rid of these "duplicated" mutants (mutants equivalent to others) in order to not count the same test requirement multiple times.

This idea of mutant redundancy is further captured by the notion of mutant subsumption [31,33,48]. A mutant M_1 subsumes a mutant M_2 if killing M_1 implies killing M_2, i.e., if fulfilling the requirement represented by M_1 means fulfilling the requirement represented by M_2. More formally, let M_1, M_2 be two observed mutants, T the universe of possible tests, and $T_1 \subseteq T$ and $T_2 \subseteq T$ the sets of tests that kill M_1 and M_2, respectively. M_1 subsumes M_2 when $T_1 \neq \emptyset$, $T_2 \neq \emptyset$, and $T_2 \subseteq T_1$.

Using the subsumption relationship, Ammann *et al.* define the minimal mutant set [1], the smallest set of mutants that subsumes the set of all analysed mutants, i.e., the set of mutants with the least redundancy. Killing all minimal mutants ensures killing all mutants, i.e., minimal mutants represent the same test requirements as all mutants, but greatly reduce the number of mutants to consider, and thus the cost of mutation analysis.

As determining true subsumption relationships is not possible (it is not possible to run all possible tests), the notion of dynamic subsumption approximates the subsumption relationship w.r.t. a given test suite. In this work, subsuming and minimal mutants are based on dynamic subsumption.

2.1.2 Regression Mutation Testing:

Applying mutation during regression testing has long been proposed. In particular, Cachia *et al.* [10] proposed applying change-based mutation testing by considering only the mutants located on the altered code. Zhang *et al.* [67] proposed Regression Mutation Testing, a technique that speeds up mutant execution on evolving systems by incrementally calculating the mutation score (and mutant status, killed/live). As such, they assume that testers should use the entire set of mutants when testing evolving software systems.

Existing mutation testing tools, such as Pitest [17], include some form of incremental analysis in order to calculate the mutation score (and mutant status, killed/live) of the entire systems or class under test. Petrovic and Ivankovic [53] use mutation within code review phase, by randomly picking some mutants located on the altered code areas.

From the above discussion it should be clear that existing techniques are either targeting the entire set of mutants or those (or some) located on the modified code areas. In the following we evaluate the appropriateness of this practice w.r.t. to changed behaviours.

2.2 Test Case Prioritisation

Testing is a key process in Software Engineering, but can also be a very expensive one. Test case prioritisation aims at ordering the different tests that are executed against the system in order to achieve some desired goal more efficiently. Tests are ordered to reveal a fault as early as possible in the execution of the test suite, providing faster feedback to testers and developers [64].

Much work has been done on test case prioritisation in the context of regression testing [27,64,29]. In this context, tests can be ordered based on their contribution to some test criterion on the previous version of the system, either totally or additionally [66]. Examples of test criteria used for regression test case prioritisation include code coverage [18], logic coverage [22], and also mutation score [60].

Mutation analysis has also been used as a metric to evaluate and compare different test prioritisation methods [27]. The different tests orderings are then compared based on how much each new test execution improves the mutation score, i.e., how fast the best possible mutation score is achieved by an ordering.

3 Commit-Relevant Mutants

Informally, a commit-aware test criterion should reflect the extent to which test suites have tested the altered program behaviours. This means that test suites should be capable of testing and making observable any interaction between the altered code and the rest of the program. We argue that mutants can capture such interactions by considering both the behavioural effects of the altered code on mutants' behaviour and visa versa. This means that mutants are relevant to a commit when their behaviour is changed by the regression changes. Indeed, changed behaviour indicates a coupling between mutants and regressions, suggesting relevance.

3.1 Rationale Behind Commit-Aware Mutation Testing

Since relevant mutants form commit-aware test requirements they should be killed by tests that exercise/test the committed code and its integration to the rest of the program. This means that relevant mutants should be killed by tests that are capable of detecting, i.e., making observable, any potential fault that depends on the commit.

To identify such mutants we check, for each mutant, whether there is at least one test case that can make observable any behavioural difference between the mutant and:

1. the program version that includes only the mutant (mutant in the pre-commit version).

2. the program version that includes only the committed changes (post-commit version).

These two conditions ensure the presence of "observable dependencies" between the mutant and the committed code since the removal of either of them impacts (changes) the behaviour of the program under the same test execution. Figure 1 illustrates the use of the above conditions. In particular, given the pre- and post-commit versions, and a mutant located on lines unmodified by the commit, denoted as pre-M and post-M, we can identify relevant mutants by checking whether there is any test case (if there exists at least one) that can make observable the differences between pre-M and post-M and between post-commit and post-M.

More formally:

- let m be a mutant of the post-commit version of the program under analysis.
- let t be a test case from a set T of all possible test cases for this program.
- let $O_v(t)$ be an execution function of a test t on a program version v. Where v takes format of:
  - *post* - the post-commit version of the program.
  - m_{post} - m mutated post-commit version of the program.
  - m_{pre} - m mutated pre-commit version of the program.
- let denote A as a set of commit non-relevant mutants.
- let denote B as a set of commit-relevant mutants.

Definition 1 Commit Non-Relevant mutant

$$m \in A := \forall (t) \in \{T\} : Om_{post}(t) = O_{post}(t) \vee Om_{post}(t) = Om_{pre}(t) \quad (1)$$

Definition 2 Commit Relevant mutant

$$m \in B := \exists (t) \in \{T\} : Om_{post}(t) \neq O_{post}(t) \wedge Om_{post}(t) \neq Om_{pre}(t) \quad (2)$$

3.2 Demonstrating Example

Figure 2 illustrates the concept of relevant mutants. The example function takes 2 arguments (integer arrays x and y of size 3), sorts them, makes some computations, and outputs an integer. The commit modification alters the statement at line *7* by changing the value assigned to the variable L from *1* to *0*, denoted with the pink-highlighted line (starting with '-') for the pre-commit version and green-highlighted line (starting with '+') for the post-commit version.

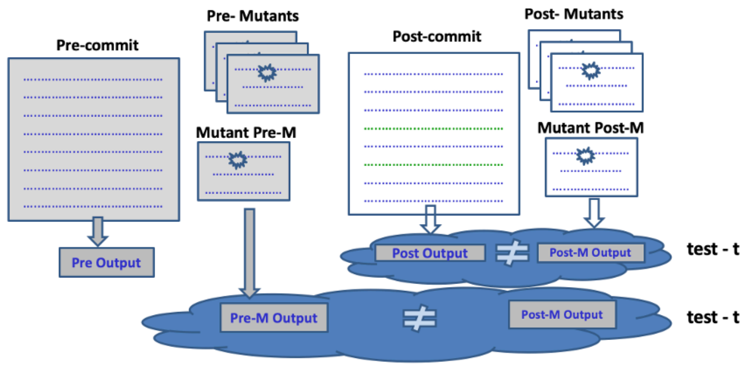
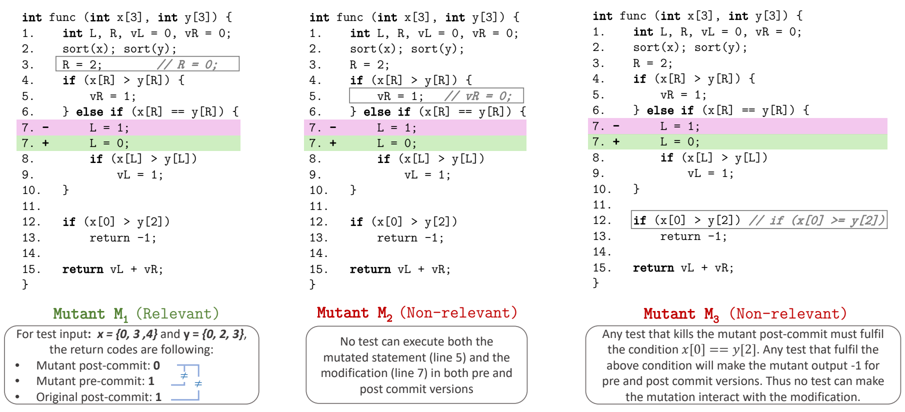


Fig. 1: A mutant is relevant if it impacts the behaviour of the committed code and the committed code impacts the behaviour of the mutant. This means that there is at least one test case (test - t) that can distinguish both the behaviours of Pre-M from Post-M and Post from Post-M.



```
int func (int x[3], int y[3]) {
1.     int L, R, vL = 0, vR = 0;
2.     sort(x); sort(y);
3.     R = 2;            // R = 0;
4.     if (x[R] > y[R]) {
5.         vR = 1;
6.     } else if (x[R] == y[R]) {
7. -       L = 1;
7. +       L = 0;
8.         if (x[L] > y[L])
9.             vL = 1;
10.    }
11.
12.    if (x[0] > y[2])
13.        return -1;
14.
15.    return vL + vR;
}
```

Mutant M_1 (Relevant)

For test input: ***x = {0, 3 ,4}*** and **y** = ***{0, 2, 3}***, the return codes are following:

- Mutant post-commit: **0**
- Mutant pre-commit: **1**
- Original post-commit: **1**

```
int func (int x[3], int y[3]) {
1.     int L, R, vL = 0, vR = 0;
2.     sort(x); sort(y);
3.     R = 2;
4.     if (x[R] > y[R]) {
5.         vR = 1;     // vR = 0;
6.     } else if (x[R] == y[R]) {
7. -       L = 1;
7. +       L = 0;
8.         if (x[L] > y[L])
9.             vL = 1;
10.    }
11.
12.    if (x[0] > y[2])
13.        return -1;
14.
15.    return vL + vR;
}
```

Mutant M_2 (Non-relevant)

No test can execute both the mutated statement (line 5) and the modification (line 7) in both pre and post commit versions

```
int func (int x[3], int y[3]) {
1.     int L, R, vL = 0, vR = 0;
2.     sort(x); sort(y);
3.     R = 2;
4.     if (x[R] > y[R]) {
5.         vR = 1;
6.     } else if (x[R] == y[R]) {
7. -       L = 1;
7. +       L = 0;
8.         if (x[L] > y[L])
9.             vL = 1;
10.    }
11.
12.    if (x[0] > y[2]) // if (x[0] >= y[2])
13.        return -1;
14.
15.    return vL + vR;
}
```

Mutant M_3 (Non-relevant)

Any test that kills the mutant post-commit must fulfil the condition $x[0] == y[2]$. Any test that fulfil the above condition will make the mutant output -1 for pre and post commit versions. Thus no test can make the mutation interact with the modification.

Fig. 2: Example of relevant and non-relevant mutants. Mutant 1 is relevant to the committed changes. Mutants 2 and 3 are not relevant.

The sub-figure on the left side shows mutant M_1. M_1 is characterised by the mutation that changes the statement $R = 2$ into $R = 0$ in line 3 (the C language style comment represents the mutant's statement). We observe that, with an input t such that $t : x = \{0, 3, 4\}, y = \{0, 2, 3\}$, the original program post-commit has an output value of 1, the mutant M_1 pre-commit outputs 1 and the mutant M_1 post-commit outputs 0. Based on the definition of relevant mutants, M_1 is relevant to the commit modification.

The sub-figure in the center shows mutant M_2 (mutation changes the statement $vR = 1$ into $vR = 0$ in line 5). We observe that the mutated statement (in line 5) and the modification (in line 7) are located in two mutually unreachable nodes of the control-flow graph. Thus, no test can execute both the changed statement and M_2. M_2 is not relevant to the commit modification.

The sub-figure on the right side shows mutant M_3 (mutation changes the expression $x[0] > y[2]$ into $x[0] >= y[2]$ in line 12). We observe that some tests execute both the commit modification and the mutated statement. However, no test can kill M_3 in the post-commit version and at the same time differentiate between the outputs of the pre-commit and post-commit versions of mutant M_3. The reason is that any test that kills M_3 in the post-commit version must fulfil the condition $x[0] == y[2]$. Any such test makes both the pre- and post-commit versions of M_3 to output -1, thus, not fulfilling the condition to be relevant. Since, there exists no such test, M_3 is not relevant to the commit modification.

Note that in case a modification inserts statements, all killable mutants (in the post-commit version) located on these statements (new statements) are relevant to the modification. In case of deletion (modifications remove statements), the mutations located on these statement do not exist in the post-commit version, and thus, are not considered.

4 Experimental Setup

4.1 Research Questions

We start our analysis by recording the prevalence of commit-relevant mutants in code commits. Thus, we ask:

RQ1: (Mutant distributions) What ratio of mutants is relevant, is located on changed code, and is located on non-changed code?

Answering this question will help us understand the extent of “noise” included in the mutation score and will provide a theoretical upper bound on the application cost of commit-aware mutation testing.

As we shall show, the majority of the mutants are irrelevant to the committed code, indicating that using all mutants is sub-optimal in terms of application cost. Perhaps more interestingly, using such an unbalanced set could result in a score metric with low precision. Therefore, we need to check the extent to which mutation score is adversely influenced by irrelevant mutants. Thus, we investigate:

RQ2: (Metrics relation) Does the mutation score (MS), computed based on all mutants, on mutants located on the committed/modified code, and the delta of the pre- and post- commit MS correlate with the relevant mutation score (rMS)?

Knowing the level of these correlations can provide evidence in support (or not) of the commit-aware assessment (i.e., the extent to which mutation score reflects the level at which the altered code has been tested). In particular, in case there is a strong correlation, we can infer that the influence of the irrelevant mutants is minor. Otherwise, the effects of the irrelevant mutants may be distorting.

While the correlations reflect the influence of the irrelevant mutants on the assessment metric, they do not say much about the extent to which irrelevant mutants can lead to tests that are relevant to the changed behaviours (in case mutants are used as test objectives). In other words, it is possible that by killing random mutants (the majority of which is irrelevant), one can also kill relevant mutants. Such a situation happens when considering the relation between mutants and faults, where mutant killing ratios have weak correlation with fault detection rates but killing mutants significantly improves fault revelation [51]. Hence we ask:

RQ3: (Test selection) To what extent does the killing of random mutants result in killing commit-relevant mutants?

We answer this question by simulating a scenario where a tester analyses mutants and kills them. Thus, we are interested in the relative differences between the relevant mutation scores when testers aim at killing relevant and random mutants. We use the random mutant selection baseline as it achieves the current best results [34, 12]. We compare here on a best effort basis, i.e., the commit-relevant mutation score achieved by putting the same level of effort, measured by the number of mutants that require analysis. Such a simulation is typical in mutation testing literature [16, 34] and aims at quantifying the benefit of one mutant selection approach over another.

Answering the above question provides evidence that killing relevant mutants yields significant advantages over the killing of random mutants. While this is important and demonstrates the potential of killing commit-relevant mutants in terms of relevance, still the question of actual test effectiveness (actual fault revelation) remains. This means that it remains unclear what the fault revelation potential of killing commit-relevant mutants is when the commit is fault-introducing. Therefore we seek to investigate:

RQ4: (Fault Revelation) How does killing commit-relevant mutants compare with killing of random mutants w.r.t. to (commit-introduced) fault revelation?

To answer this question we investigate the fault revelation potential of killing commit-relevant mutants based on a set of real fault-introducing commits. We follow the same procedure as in the previous research question (RQ3) in order to perform a best effort evaluation. While answering the question about mutant's fault revelation ability, and showing their usefulness and practicality in finding faults, we would like to know whether commit-aware mutants can be found through different classes of mutants. If the majority of the relevant mutants are also part of other mutant classes, it indicates that the other classes can be used as a proxy to relevant mutants. This is important since previous research [55, 66, 19, 27] heavily relied on other mutant classes to evaluate regression testing techniques. Moreover, by investigating the relationship with other mutant classes we can better understand the nature of relevant mutants and their fundamental differences (and similarities) with other classes. In particular, by investigating the relationship with Subsuming mutants, we can see how many subsuming mutants are relevant, which represents the relevant behaviours captured by the mutants over all behaviours, i.e., ratio of relevant over all mutants after minimising the noise from redundant mutants. This allows us to have a better understanding of the discrepancies and potential wasted effort caused by irrelevant mutants. Similarly, the comparison with hard-to-kill mutants can show whether relevant mutants are not that difficult to kill and somehow distinct from the other classes. Thus, we are engaged in knowing:

RQ5: (Mutants Classes) How different the relevant mutants are to the classes of subsuming and hard-to-kill mutants?

We answer this question by investigating the relationship and overlap among the three sets of relevant mutants, hard-to-kill mutants, and subsuming mutants. As a reminder for our reader, hard-to-kill mutants are the set of mutants killed by few tests.

Overall, answering the above questions will improve the understanding of the potential of the cost-effectiveness application of commit-aware mutation testing.

4.2 Analysis Procedure

We performed mutation testing on the selected subject using all the mutation operators supported by Mart [14] and Pitest [17] (the mutation testing tools we use). For the C programs, before conducting and running any experiment we discarded all the trivially equivalent mutants (including the duplicated ones), using the TCE method [30,49] in order to reduce their impact on our results. This is an important step in order to avoid influence from trivially equivalent mutants that could anyway be reduced using the TCE method. Therefore, we applied our analysis on the resulting sets of mutants i.e., those that are not trivially equivalent.

Identifying relevant mutants requires excessive manual analysis, thus we approximate them based on test suites (this is a typical experimental procedure [1,34,50]). To do so we composed large test pools, which approximate the input domain. The pools are composed of the post-commit version developer tests (mined from the related repository). For C programs we augment the pools with automatically generated tests, similarly to the process followed by Kurtz *et al.* [34] and Papadakis *et al.* [50].

Using the test pools, we execute all the mutants (on both pre- and post-commit versions) and construct the mutation matrix that records the test execution output of each test on each mutant. For C programs, the output is the standard output produced when running the test, while for the Java programs it is the status (pass/fail) of the test run.

By using this information, i.e., test execution outputs on every related version, we approximate the relevant mutant set based on Algorithm 1. In the algorithm, the function calls *postCommitOrigOutput*, *postCommitMutOutput* and *preCommitMutOutput* compute the output of the execution of test case 'test' on the post-commit original program, post-commit version of mutant 'mut' and pre-commit version of mutant 'mut', respectively. In particular, in C the function calls *postCommitOrigOutput*, *postCommitMutOutput* and *preCommitMutOutput* return the exact concrete value of a test case output when executed on a mutant or the original program. While in Java they provide standard unit-level oracle pass/fail output. The generated output of the execution of test case 'test', on each version of a software (post-commit version, mutant M generated on the pre- and post-commit version) is compared between the versions to identify a difference in behavior between mutant pre-M and post-M.

Besides the relevant mutant set, we also extract the modification mutant set, made of mutants that are located on a statements modified or added by the commits. This set is computed by extracting the modified or added statements from the commit *diff* and collecting the mutants that mutate those statements. Note that, by definition, the killable modification mutants are also relevant mutants, as their pre-commit output is not defined, and thus different from their post-commit output.

Therefore, we have a set of all the mutants generated on a post-commit version of a program (post-M), which can be divided into two subsets; those that are identified as commit-relevant by our approach (commit-relevant) and those that are identified as commit-irrelevant (non-relevant). The set of post-commit mutants located on statements modified or added by regression changes forms the (modification) subset of mutants. As already mentioned the modification set of mutants includes both relevant and irrelevant mutants. In RQ2, we want to know the correlations between the mutation scores of the aforementioned mutant sets. To do so, we select arbitrary test sets of various sizes and record the mutation scores on each mutant set and compute their correlations.

Algorithm 1: Approximate Relevant Mutants Set

```
Data: TestSuite, Mutants
Result: Relevant Mutants
$RelevantMuts \leftarrow \emptyset$;
for $mut \in Mutants$ do
    for $test \in TestSuite$ do
        $origV2 \leftarrow postCommitOrigOutput(test)$;
        $mutV2 \leftarrow postCommitMutOutput(test, mut)$;
        $mutV1 \leftarrow preCommitMutOutput(test, mut)$;
        if $origV2 \neq mutV2 \wedge mutV2 \neq mutV1$ then
            $RelevantMuts \leftarrow RelevantMuts \cup \{mut\}$;
            break;
        end
    end
end
return $RelevantMuts$ ;
```

In RQ2 we arbitrary pick sets of tests representing 10%, 20%, ..., 90% of the test pool. As these sets are randomly sampled we selected multiple sets (500 for C and 100 for Java) per size considered and per program commit (each subset of test can be seen as a testing scenario). For every test set, we computed the mutation score for each of the three mutant sets. We name as *MS*, *rMS* and *mMS* the mutation scores for the whole mutant set, relevant mutant set and modification mutant set, respectively. The mutation scores are computed on the post-commit versions and using the mutation matrix. Thus, for each commit and each test size, we have three statistical variables (*MS*, *rMS* and *mMS*), whose instances are the corresponding mutation scores for each test set.

Having collected the data for the statistical variables *MS*, *rMS* and *mMS*, we compute the correlations between *rMS* and *MS* as well as the correlation between *rMS* and *mMS*. If the correlation between *rMS* and *MS* (*mMS*) is high, it means that *MS* (*mMS*) can be used as a proxy fo *rMS*. Otherwise, *MS* (*mMS*) is not a good proxy for *rMS* and thus, *rMS* should be targeted directly.

We also computed, for each test set, the mutation score in the pre-commit version. Then we compute the absolute change of mutation score (named *deltaMS*), on the analyzed mutant set, incurred by a commit modification ($delatMS = |MS_{post-commit} - MS_{pre-commit}|$), and we compute the correlation between *rMS* and *deltaMS*. A strong correlation would mean that the absolute change of mutation score between versions is a proxy for *rMS*. Weak correlation would mean that *rMS* cannot be represented by *delatMS*.

In RQ3, we simulate a scenario where a tester selects mutants and designs tests to kill them. This is a typical evaluation procedure [34, 50] where a test that kills a randomly selected mutant (from the studied mutant set) is selected from the test pool. This test is then used to determine the killed mutants, which are discarded from the studied mutant set. The process continues (by picking the next live mutant) until all mutants have been killed. If a mutant is not killed by any of the tests, we treat it as equivalent. This means that our effort measure is the number of mutants picked (either killable or not) and effectiveness measure is the relevant mutation score. Since we perform a best-effort evaluation we focus on the initial few mutants (up to 50) that the tester should analyse in order to test the commits under test. We repeat this process (killing all mutants) 100 times and compute the relevant mutation score.

For RQ4, we repeat the same procedure as in RQ3. However, instead of computing the relevant mutation score, we compute the fault revelation probability.

For RQ5, we calculate the overlap of three different categories of mutants: relevant mutants, hard-to-kill mutants, and subsuming mutants. We compute the set of subsuming mutants following the standard subsumption theory described in Section 2.1.1. As typically performed, we use the available tests to compute the subsumption relationships [50, 34]. Based on these relationships, we determine the subsuming mutants set. When it comes to the set of hard-to-kill mutants, we consider as hard-to-kill any mutant killed by less than 2.5%

of covering tests, i.e., a mutant M is hard-to-kill if and only if less than 2.5% of the tests that cover M also kill M. For analysis of our results, we calculate relation between the corresponding sets of mutants, following percentage formula: sample/population $\times$ 100.

4.3 Statistical Analysis

We perform a correlation analysis to evaluate whether the mutation score, when considering all mutants, correlates with the relevant mutation score. To this end, we use two correlation metrics: *Kendall rank coefficient (τ)* (Tau-a) and *Pearson product-moment correlation coefficient (r)*. In all cases, we considered the 0.05 significance level.

The Kendall rank coefficient τ, measures the similarity in the ordering of the studied scores. We measure the mutation score MS and the relevant mutation score rMS when using test suites of size 10%, ..., 90% of the test pools. The Pearson product-moment correlation coefficient (r) measures the covariance between the MS and rMS values. These two coefficients take values from -1 to 1. A coefficient of 1, or -1, indicates a perfect correlation while a zero coefficient denotes the total absence of correlation.

To evaluate whether the achieved mutation scores MS and relevant mutation scores rMS are significantly different, we use a Mann-Whitney U Test performed at the 0.05 significance level. This statistical test yields a probability called p-value which represents the probability that the MSs and rMS are equal. Thus, a p-value lower than 0.05 indicates that the two metrics are statistically different. We use paired and two-tailed U test, to account for the different commits and programs.

4.4 Program Versions Used

To answer RQs 1-3 we used the C programs of GNU Coreutils[1], used in many existing studies [32, 13, 11]. GNU Coreutils is a collection of text, file, and shell utility programs widely used in Unix systems. The whole code-base of Coreutils is made of approximately 60,000 lines of C code[2]. In order to obtain a commit benchmark of Coreutils programs we used to following procedure to mine recent commits from the Coreutils github repository. (1) We set the commit date interval from year 2012 to 2019. This resulted in 5,000 commits considered. (2) Next, we filtered out the commits that do not alter source code files. This resulted in 1,869 commit remaining. (3) Then, we only kept the commits that affect only the main source file of a single program (This enable better control of test execution, because other programs of Coreutils are often used to setup the test execution of a tested program). (4) After that,

[1] https://www.gnu.org/software/coreutils/
[2] Measured with cloc (http://cloc.sourceforge.net/)

Table 1: C Test Subjects

Benchmark	#Programs	#Commits	# Mutants	#Test cases
CoREBench [7]	6	13	154,396	8,828
Benchmark-1	13	34	338,390	11,866

we filtered out commits that are very large (commits whose modification has an edit actions of more than 5 according to GumTree [21]). This resulted in 218 commits. (5) Due to the large execution time of the experiments, approx. 2 weeks of CPU time per commit, we randomly sampled 34 commits among the remaining commits for the experiments. This constitutes our Benchmark-1.

In order to further strengthen our experiment and answer RQ4, we also use 13 commits from the CoREBench [7] that introduce faults. We selected these commits to validate the fault revelation ability of relevant mutants. Since we approximate relevant mutants, we needed commits where automated tests generation frameworks could run. Thus, we limit ourselves to the 18 fault introducing commits of Coreutils that we can run with Shadow symbolic execution [32]. Among these faults, two were discarded due to technical difficulties in compiling the code (the build system uses very old versions of the build tools). Three faults were discarded due to the excessively high required execution time to run the mutants (we stopped after 45 days).

Table 1 summarizes the information about the C language benchmarks used in the experiments.

To answer RQs 1-3, we also consider a set of commits from well-known and well-tested Java programs. We extract these commits from projects in the Apache Commons Proper repository[3], a set of reusable Java component projects, from Joda Time[4], a time and date library, and Jsoup[5], an HTML manipulation library. For each of the projects, we manually gathered the most recent commits meeting the following conditions from the project's history: (1) only source code is modified, no modification to configuration files, (2) the commit introduces a significant change, not a trivial one such as a typo fix, (3) test contracts are not modified, in order to meaningfully compare pre- and post-commit outputs and (4) both pre- and post-commit versions of the project build successfully. Overall, we gathered 36 commits, Table 2 summarises information about the commits used from each project.

4.5 Mutation Mapping Across Versions

As mutation testing tools generate mutants for a given program version instead of regression pairs, we need to identify the common mutants between the two

[3] `https://commons.apache.org/`

[4] `https://github.com/JodaOrg/joda-time/`

[5] `https://github.com/jhy/jsoup`

Table 2: Java Test Subjects

Project	**# Commits**	**# Mutants**	**# Test cases**
commons-cli	9	61,419	3,247
commons-collections	5	323,584	55,076
commons-io	3	105,181	3,972
commons-net	6	345,130	1,478
joda-time	5	561,782	20,962
jsoup	8	330,125	4,985

versions. In other words, we need to map each mutant from its pre- to post-commit version of the program.

To establish such a mapping in the case of C programs, we unify the commit modifications into a single program, as done in the literature [32], and apply any standard (unmodified) mutation tool to generate the mutants. The code unification of the commit modification is done through annotation that has no side-effect. The annotations are made through a special function called *"change"* that takes 2 arguments/values (the arguments are the value of the pre-commit and post-commit versions, respectively) and return one of the two values.

The annotations are manually inserted in the program, according the semantics presented in previous studies [32].

Note that the statement insertion can be annotated by wrapping the inserted statement with $if(change(false, true))$; and a statement deletion can be annotated by wrapping the deleted statement with $if(change(true, false))$.

The choice of the version to use, for each mutant, is decided at runtime (by specifying the version to use through an environment variable recognizable by the *change* function).

For the Java programs, we perform the mapping of mutants from both sets of mutants of pre- and post- commit versions and the commit diff. First we start by generating the mutants for both pre- and post-commit versions of the program using the mutation tool. We then map pre- and post- commit line numbers by parsing the commit diff, such as that we can identify which lines have been altered between the versions. Then, we use this mapping of altered line numbers to map pre- and post-commit mutants: using the line number, bytecode instruction number and mutation operator of the mutants to match both sets. We adopt this way for the Java programs in order to avoid making drastic changes on Pitest (the mutation testing tool we use).

4.6 Mutation Testing Tools and Operators

As test suites are needed in our experiment, we use the developer tests suites for all the projects that we studied. These were approximately 4,194 tests in total for C programs.

To strengthen the test suites used in our study, we augment them in two phases. First, we use KLEE [11], with a robust timeout of 2 hours, to perform a form of differential testing [20] called shadow symbolic execution [32], which generates 234 test cases. Shadow symbolic execution generates tests that exercise the behavioural differences between two different versions of a program, in our case the pre-commit and the post-commit program versions.

In order to also expose behavioural difference between the original program and the mutants, we used SEMu [13], with a robust timeout of 2 hours, to perform test generation to kill mutants in the post-commit program versions. SEMu generates 17,915 test cases.

These procedures resulted in large test suites of 22,343 test cases for C programs in total. Since we compare program versions, we use the programs output as an oracle. Thus, we consider as distinguished or killed, every mutant that results in different observable output than the original program.

We use Mart [14], a mutation testing tool that operates on LLVM bitcode, to generate mutants. Mart implements 18 operators (including those supported by modern mutation testing tools), composed of 816 transformation rules.

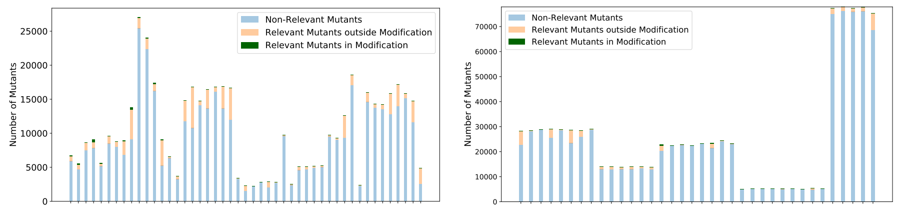
To reduce the influence of redundant and equivalent mutants, we enabled Trivial Compiler Equivalence (TCE) [30, 49, 26] in Mart to detect and remove TCE equivalent and duplicate mutants.TCE detected 13,322 and 460,072 equivalent and redundant mutants.

For the Java programs, we use the developer test suites available. We perform mutation analysis using Pitest[17], a state of the the art mutation testing tool that mutates JVM bytecode. We use all mutation operators available in Pitest, which are described in [36].

5 Results

5.1 RQ1: Relevant mutant distribution

We start our analysis by examining the prevalence of commit-relevant mutants, i.e., mutants that affect the altered program behaviours. Figure 3 records the distribution of the relevant and non-relevant mutants among the studied commits. Based on these results we see that only a small portion of the mutant population produced by the selected mutation operators is actually relevant. This portion ranges from 0.5% to 47%, among which 3.6% is located on the changed program lines, while the rest is located on the rest of the code. For the large portion, it is possible to happen when the source code is not large, and the change is located in the crucial position.



(a) C programs (b) Java programs

Fig. 3: The distribution of killable, non-relevant, relevant outside the modification and relevant on the modification mutants among the studied commits.

Interestingly, the presence of so many "irrelevant" mutants, can have major consequences when performing mutation testing. Such consequences are a distorting effect on the accuracy of the mutation score, and a waste of resources when executing and trying to kill non-relevant to the commit mutants. We further investigate these two points in the following sections.

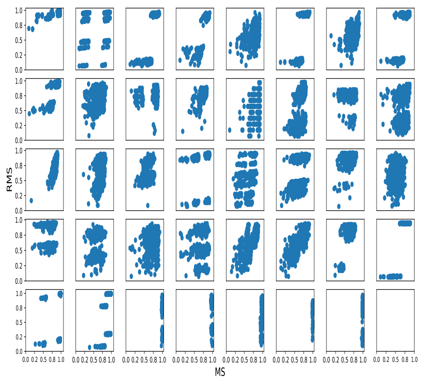
5.2 RQ2: Relevant mutants and mutation score

Figure 4 visualizes our data; each data point represents the mutation score and relevant mutation score of a selected test suite. As can be seen from the scatter plots, there is no visible pattern or trend among the data. We can also see that there is a large variation between mutation scores and relevant mutants scores in almost all the cases. These observations indicate that the examined variables differ significantly. In other words, one cannot predict/infer one variable using the other one. To further explore the relationship between mutation score and relevant mutation score within our data we perform statistical correlation analysis.

Finding a strong correlation would suggest that the two metrics have similar behaviours (an increase or decrease of one implies a relatively similar increase or decrease of the other). Figure 5 displays the results for the two correlation coefficients that have statistically significant values for randomly selected test suites (from our test suite pool) of different sizes[6]. Interestingly, we observe that most of the correlations are relatively weak with their majority ranging from 0.15 to 0.35. Additionally, we see that both coefficients we examine are aligned, indicating a weak relationship when either ordering test suites or considering their score differences.

One may assume that the relevant mutation score may be well approximated by the mutants that are located on the modified code, assuming that mutants' location reflects their utility and relevance. Similarly, one may assume that the commit-relevant score could be approximated by the delta of the pre- and post-commit mutation scores. We investigate these cases and find

[6] We observe similar trends with Pearson correlation. Due to lack of space, Pearson correlation results can be found in the accompanying website.


(a) C programs

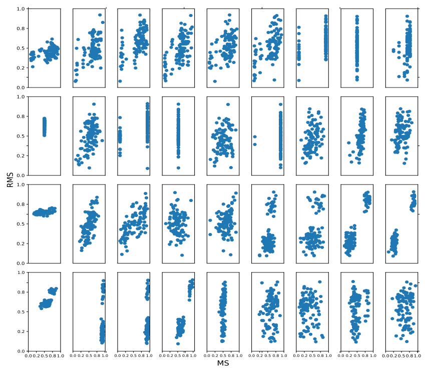
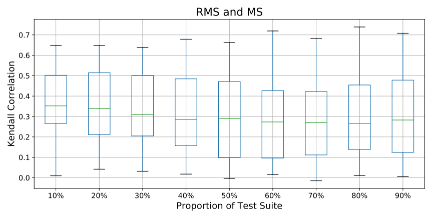

(b) Java programs

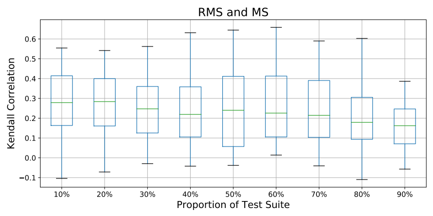
Fig. 4: The relationship between Mutation Score and Relevant Mutation Score.

that most of the correlations are relatively weak with their majority ranging from -0.1 to 0.1.

Overall, our results indicate that irrelevant mutants have a major influence on the mutation score calculation, and that using the overall mutation score does not reflect the actual value of interest, i.e., how well the altered behaviours are tested, which is represented by relevant mutation score (rMS). Approximating the rMS using either the deltaMS or the mutants of the altered lines is also not sufficient. Hence, our results suggest that MS and other direct metrics are not good indicators of commit-related test effectiveness. We envision that future research should develop techniques capable of identifying relevant mutants at testing time, i.e., prior to any test generation and mutant analysis, in order to support testers.

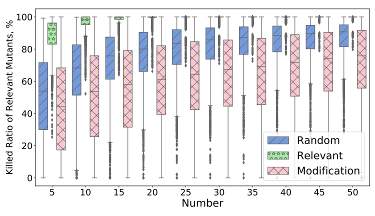


(a) C programs

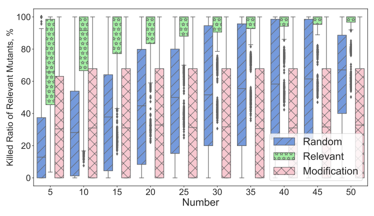


(b) Java programs

Fig. 5: Correlation between Mutation Score and Relevant Mutation Score for different test suite sizes on different languages.



(a) C programs



(b) Java programs

Fig. 6: Test suite improvement of mutation-based testing with random (traditional mutation) and relevant mutants.

Table 3: $\hat{A}_{12}$. rMS when aiming at Relevant, Random and Modification related mutants.

#Mutants	**5**	**10**	**20**	**30**	**40**	**50**
Relevant-Random	0.90	0.95	0.98	0.98	0.98	0.97
Relevant-Modification	0.89	0.96	0.99	0.99	0.99	0.99

5.3 RQ3: Test Selection

Recent research has shown that mutation testing is particularly effective at improving test suites and revealing faults (guiding testers to design test cases that reveal faults), while at the same time mutation score is weakly correlated with fault detection [51]. In view of this, it is possible that despite the weak correlations we observe in our case, traditional mutation could successfully guide testers towards designing tests that collaterally kill relevant mutants.

Results are recorded in Figure 6 for the first 1-50 mutants to be analysed by the tester. We observe a large divergence (approximately 50%-60%) between the random, commit-based and relevant mutants. This suggests that by analyzing mutants in interval of 5, from selected 5 mutants to selected 50 mutants at random, one would miss approximately 60% and 50% of commit-relevant mutants for C and Java programming languages, respectively. This difference is statistically significant and with large effect size (Effect Size values are recorded on Table 3). Moreover, what we can observe that as we start increasing the number of analyzed mutants (5-50 mutants), the difference between the killed ratio of relevant mutants decreases. This is expected since putting more effort essentially results in selecting more mutants thereby increasing the chances to select some relevant. Taking together the weak correlations we found in the previous section with these results, we conclude that traditional mutation testing is sub-optimal and cannot be used to assess or guide (in a best-effort basis) the testing of committed code. Therefore, to support practitioners, future research should aim at identifying and using commit-relevant mutants. Similarly, controlled experiments should be based on relevant mutants when aiming at assessing change-aware test effectiveness.

Table 4: $\hat{A}_{12}$. Fault revelation when aiming at Relevant, Random and Modification related mutants.

% Relevant mutants analysed	**10%**	**20%**	**50%**	**75%**	**100%**
Relevant-Random	0.55	0.59	0.64	0.66	0.64
Relevant-Modification	0.57	0.59	0.69	0.73	0.70

5.4 RQ4: Fault Revelation

To demonstrate the importance of commit-aware mutation testing, we further compare the ability of the traditional mutants and commit-relevant mutants to reveal commit-introduced faults (real faults). We follow the same procedure as in the previous section but evaluate w.r.t. to the rate of faults revealed by the selected test suites.

The fault revelation results are depicted in Figure 7. From this data, we can see that a significant fault revelation difference (approximately 30-40%) between the compared approaches can be recorded. This difference is statistically significant with large effect size (Effect Size values are recorded on Table 4). Here it must be noted that these results can be achieved by an effort equivalent to analysing 0.4% of the mutants, which is 27 mutants per commit (on average).

Overall, our results demonstrate that by aiming at relevant mutants one can achieve significant fault revelation benefits (approximately 30%) over the traditional way of using mutation testing.

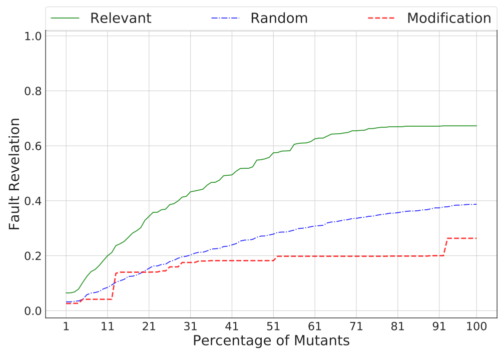


Fig. 7: Fault revelation of mutation-based testing with random (traditional mutation) and relevant mutants.

5.5 RQ5: Mutant Classes

Figure 8 shows the overlap among the relevant, subsuming, and hard-to-kill mutant classes for the C and Java benchmarks. From these results we can conclude:

Commit-Relevant vs. Subsuming: Our results show that most of the relevant mutants are also non-subsuming, more precisely 59.79% and 84.45% (11.03 / (11.03+0.2+0.23+1.6) * 100) for both C and Java benchmark. Suggesting that relevant mutants have many redundancies, similarly to other mutants. Now, if we measure overlapping just between those two categories, commit relevant and subsuming mutants, we see an overlap of 11.38% and 0.21% for both languages, respectively. This overlap is small implying an imbalanced case, i.e., by targeting subsuming mutants one wastes significant resources than if targeting commit relevant mutants.

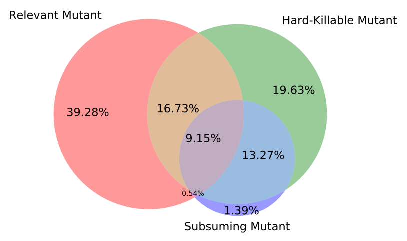
An interesting finding here is that most of the commit-relevant mutants are also non-subsuming, meaning that relevant mutants have many redundancies, similarly to other mutant classes. This is important since it indicates that mutant selection may also benefit and be improved by emerging work on subsuming mutant selection [44, 25, 24].

Commit-Relevant vs. Hard-To-Kill: The results of the comparison with Hard-to-kill mutants shows that relevant mutants are not that difficult to kill and somehow distinct from the other categories. Among the union of mutants, 19.63% and 2.69% represent hard-to-kill mutants that do not fall under other classes, both for C and Java, respectively. More precisely, 33.39% and 32.84% of mutants from the hard-to-kill set do not overlap with other classes, while we can observe the overlap of 17.06% and 1.12% for both languages. The overlap is small because the committed changes are in relevantly easy-to-reach points of the programs.

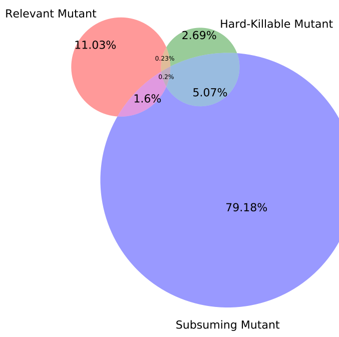
In conclusion, relevant mutants is a distinct class of mutants that is hardly approximated by other mutant classes. This means that if one would like to use mutation testing to assess change-relevant fault detection (simulating faults introduced by commits), will need to rely on relevant mutants since any other form is inherently different.

6 ShowCase the use of Relevant Mutants in assessing Regression Test Prioritisation methods

The primary purpose of regression testing is to validate whether the changes performed to the software break any of the unchanged program functionality. Therefore, test prioritization is used to support the regression testing of commits, giving rise to the question of how well they perform against commit-relevant faults. We, therefore, use commit-relevant mutant score as a metric to evaluate the ability of test prioritization techniques to detect change-relevant



(a) C programs



(b) Test Java programs

Fig. 8: Relevant Mutants intersection with Subsuming and Hard-to-Kill Mutants

faults. Our motivation is to showcase the use of relevant mutants in a regression scenario where mutants serve as a proxy for the introduced faults (from the commits) and use them to assess test case prioritization approaches, i.e., assess how well test case prioritization techniques reveal faults introduced by the commits. This is important since previous research [55,66,19,27] heavily relied on other mutant classes to evaluate regression testing techniques.

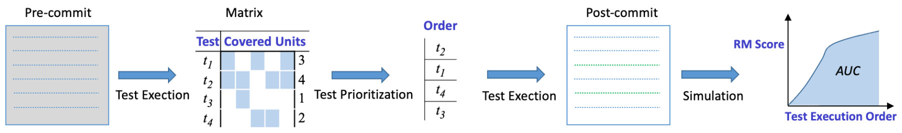


Fig. 9: Test Prioritisation Pipeline

To demonstrate the use of commit-relevant mutants, we illustrate their application in evaluating regression test case prioritisation techniques. We thus, apply popular test case prioritisation techniques to the commits that we used in our experiments and evaluate their performance w.r.t commit-relevant mutation score. In particular, for every commit, we collect the statement-, branch-, and mutant-coverage information of the available tests (the same tests used in Section 4) on the pre-commit version of the programs. Each test is then run in isolation and produces a trace of units (statements, branches, mutants) covered. These traces direct the different test prioritisation methods, which are listed in Table 5. This study considers the most popular priorization techniques, i.e., total incremental coverage test prioritisation methods [66], as described in the related literature [56].

On the post-commit version of the programs, we execute different test orderings generated from each pre-commit coverage. Each new test, executed following the ordering, may kill new commit-relevant mutants and thus increase the commit-relevant mutation score. This evolution of the relevant mutation score along the test ordering is recorded and represent a curve. The area under that curve divided by the total number of test cases represents the average Average Percentage of Fault Detected (APFD) [56]. Note that in our context, the (artificial) faults are the commit-relevant mutants. The APFD shows the average commit-relevant mutation score achieved across all possible numbers of tests, taken according to the orderings. The APFD shows how well the ordering prioritises tests killing commit-relevant mutants, i.e., tests that are relevant to the commit. The higher the APFD, the more commit-relevant tests are prioritised.

Figure 9 visualises the process of the use case we conduct.

Table 5: Test Prioritisation Criteria

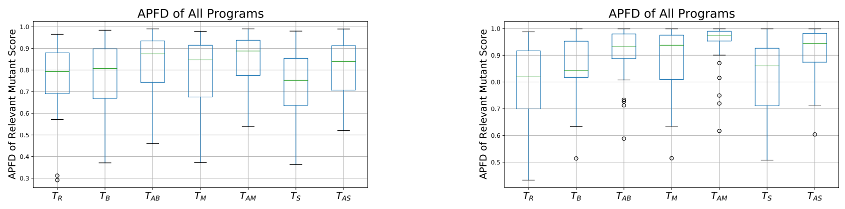
Acronym	**Name**	**Prioritisation Objective**
T_R	*Random*	Cover by randomised ordering
T_B	*Total Branch*	Cover the maximum number of branches
T_{AB}	*Additional Branch*	Cover the maximum number of uncovered branches
T_M	*Total Mutant*	Cover the maximum number of killed mutants
T_{AM}	*Additional Mutant*	Cover the maximum number of mutants not yet killed
T_S	*Total Statement*	Cover the maximum number of statements
T_{AS}	*Additional Statement*	Cover the maximum number of uncovered statements

Figure 10 shows the experimental results, aggregated across commits for both the C (Figure 10(a)) and Java (Figure 10(b)) benchmarks. For each test prioritisation technique (see Table 5), it records the APFD values achieved by the method.

The results on C programs, shown in Figure 10(a), do not indicate significant benefits from the total coverage methods (T_S, T_B, T_M) over random selection (T_R). Additional coverage methods (T_{AM},T_{AS}) result in higher APFD values, showing that the test orderings produced by these methods better detect commit-relevant mutants. However, the improvements are relatively small, indicating that further research, perhaps change-aware test prioritisation should be considered when testing such cases.

The results on Java programs, shown in Figure 10(b), however, show clear benefits from all coverage-based test prioritisation techniques over the random test prioritisation, as well as more difference between the different criteria. Mutation-based prioritisation performs best in terms of the APFD values, while statement-based prioritisation performs the worst. Similar to the results shown for the C programs, additional coverage based methods perform better than total coverage based methods. Additional mutant coverage prioritisation T_{AM} performs best, achieving over 0.9 APFD for most commits.

Furthermore, what is also interesting to observe in the box plot is the high APFD value for τ_{AM}. It indicates that mutation-based prioritisation remains robust in the presence of the committed changes.



(a) C programs (b) Java programs

Fig. 10: Test Prioritisation.

Overall, we have shown that the approaches featuring the "total" strategy perform worst, in contrast to the additional strategy which offers more robust test prioritisation. This conclusion conforms with the one in [27], which shows that the best approaches reach the APFD median of approximately equal to 87%. In our case, our best additional approaches reach APFD median of approximately 95%, while the best approach T_{AM}, reaches the average value above 95% for 75% of commits and above 90% for 100% of commits. One key insight out of the above is that test prioritisation offers relatively small improvements, indicating that further research, should be directed towards change-aware test prioritisation.

7 Discussion

7.1 Difference between C and Java programs and mutants

We experimented with programs written in both Java and C programming languages. Naturally, we observe some differences between these two languages. This is because we use different mutation testing tools, i.e., in C we use Mart, a tool that operates on LLVM bitcode, while in Java we used Pitest, which reflect properties of the underlying languages. To this end, the number of mutants generated in Java is considerably higher than that of C, but at the same time, the committed changes are smaller, as a ratio of lines of code changed to a total number of lines of code involved (due to program's size). This discrepancy results in significantly more subsuming mutants than commit-relevant in Java than in C. Nevertheless, our key conclusion regards the difference between the two sets of mutants which is significant in both languages we study.

7.2 Analysis of commit relevant mutants

Our relevance definition include the set of mutants that can be impacted by the committed changes by at least one test case. Strictly speaking this definition allows the inclusion of mutants that may be killed by tests irrelevant to the committed code. Though, we consider them as interesting as these tests exercise code parts, the parts where these mutants are located, that depend on the changed introduced by the commits. Therefore, these tests indirectly exercise the committed code. In view of this, one can define different levels of relevance by considering the strength of the dependence between the mutants and the commits. We can thus define a strong relevance relationship by mandating an observable difference between pre-M and post-M by every test case that kills mutant M. In such a case we can define a weak relevance relationship, w.r.t., complete relevance relationship, as we do in this paper, by mandating an observable difference between pre-M and post-M by at least one test case that kills mutant M.

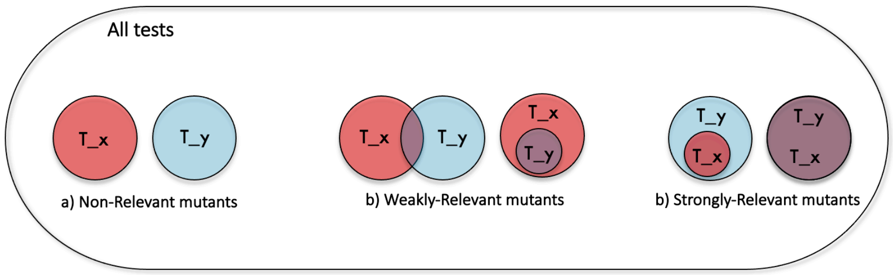


Fig. 11: Illustration of different levels of relevance. The outer rounded rectangle represents all tests of the program under test. Set of tests T_x (red circle) includes all tests t that make observable the differences between post-commit and mutated post-commit version of a program under test (Definition 2). Set of tests T_y (blue circle) includes all tests t that make observable the differences between mutated post-commit version and mutated pre-commit version (Definition 2). We identify three cases, a) Non-Relevant mutants, i.e., no test t belongs to both T_x and T_y, b) weakly-relevant mutants case with least one test t that belongs to T_x and not to T_y, c) strongly-relevant mutants where every test t that satisfies T_x also satisfies T_y.

Formally:

- let m be a mutant of the post-commit version of the program under analysis.
- let t be a test case from a set T' of all test cases that kill mutant m.
- let $O_v(t)$ be an execution function of a test t on a program version v. Where v takes format of:

  - m_{post} - m mutated the post-commit version of the program.
  - m_{pre} - m mutated the pre-commit version of the program.
- let denote D as a set of strong commit relevant mutants.

Definition 3 Strong commit relevant mutant

$$m \in D := \forall\,(t) \in \{T'\} : Om_{post}(t) \neq Om_{pre}(t) \tag{3}$$

This definition allows selecting mutants that always lead to commit-relevant tests, i.e., tests that directly exercise the committed code. Informally, the mutant relevance leans on the strength of dependency between a mutant and commit changes, expressed through the number of tests that observe the dependency over the number of tests that kill the mutant. The value of relevance lies between 0 and 1. When relevance takes value 0 there is no observable behavioural difference on test outputs impacted by code-change. As the value of relevance increases, the mutant is assessed to be more relevant, until the value equals 1, in which case the behavioural difference can be observed with every test, making a mutant strongly relevant.

Formally, let's consider the same notations as for the definition of strong commit relevant mutants. Let t be a test case from a set T of all possible test cases for a program under analysis. Thus, formal definition of mutant relevance level can be defined as follows:

Definition 4 Relevance

$$relevance(m) = \frac{|\forall\,(t) \in T : t \in T' \land Om_{post}(t) \neq Om_{pre}(t)|}{|T'|} \tag{4}$$

Furthermore, in RQ5 (Mutants Classes) we witnessed that most of the relevant mutants are non-subsuming. This phenomenon suggests that many relevant mutants are redundant. This means that one could envision an optimised scenario where redundancies among relevant mutants are minimised. Thus, we can define subsuming commit relevant mutants, which is the set of relevant mutants that subsumes the set of all relevant mutants. Formally, let M be a set of mutants $\{m_0, ..., m_n\}$ for post-commit version of the program under test. Let RM be a set of commit-relevant mutants, whereas $RM \subset M$. Let $subsuming(M)$ be a function that returns the subsuming mutant subset of M [28,1]. Thus, the set of subsuming commit-relevant mutants SRM can be defined as:

Definition 5 Subsuming commit-relevant mutants

$$SRM = subsuming(RM) : SRM \subset RM \subset M \tag{5}$$

8 Threats to validity

External validity: We selected commits that do not modify test contracts. Such commits are common in industrial CI pipelines [37] but rare in open source projects. To mitigate this threat, we performed our analysis on a relatively large set of commits given the computational limits posed by mutation analysis. In C, our experiment required on average approximately 2 weeks of CPU time to complete, per commit studied (executions performed using Muteria [15]). In addition, we used an established research benchmark (CoREBench [7]) where we found similar results. Unfortunately, we consider fault introducing commits only in C as the Java datasets do not adhere to our non-changed test contract requirement.

Another threat may relate to the mutants we use. To reduce this concern we used a variety of operators covering the most frequently used language features including the operators adopted by the modern tools [36], in both C and Java.

Another threat may relate to the occurrence of flaky tests. We believe that we bypassed this threat by running 5 times all test cases of each project and its corresponding version. However, we consider more than one reason why flaky tests should not change conclusions related to our results. First, we worked with open-source software that does not contain solid environmental dependencies, one of the leading root causes of flakiness [35]. Second, all the programs we used as a benchmark for our study are well-studied projects with a reliable test suite with no previous reports on the occurrence of flaky tests. And as third, we consider that we study versatile and various projects for both C and Java programming languages. Thus, we have reduced any potential external validity related to the flaky tests.

Internal validity: Such threats lie in the use of automated tools, the way we treated live mutants and non-adequate test suites. To diminish these concerns, we used KLEE, a state of the art test generation tool and strong mature developer test suites. Nevertheless, the current state of practice [53] relies on non-adequate test suites, so our results should be relevant to at least a similar level of practice. To ensure our results, we carefully checked our implementation and performed a manual evaluation on a sample of our results. Moreover, we use established tools also employed by numerous studies.

To deal with randomness and minimize stochastic effects, we repeated our experiments 100 times and used standard statistical tests and correlations.

Construct validity: Our effort related measurement, number of analysed mutants, essentially captures the manual effort involved in test generation. Automated tools may reduce this effort and change our best-effort results. Still, we used the current standards, i.e., TCE [30] to remove all trivially equivalent mutants before conducting any experiment and KLEE (including a mutation-based test generation approach [13]). In test generation, we acknowledge that automated tools may generate test inputs that kill mutants, but we note that they fail to generate test oracles. Therefore, even if such tools are used, the test oracles will still require human intervention, i.e., introduce some effort. Here it should be noted that we consider the mutant execution cost as negligible since it is machine time and our focus is on the human time involved when performing mutant analysis. Moreover, existing advances [65] promises to reduce this cost to a practically negligible level.

Overall, we believe that our effort measurements approximate well (in relative terms) the human effort involved. All in all, we aimed at minimizing potential threats by using various metrics, well-known tools and benchmarks, real and artificial faults and following methodological guidelines [50]. Additionally, to enable reproducibility and replication we make our tools and data publicly available[7].

9 Related Work

There are various methods aiming at identifying relevant coverage-based test requirements in the literature. For instance, it has been proposed to consider as relevant every test element that can be affected by the changes (by doing some form of slicing, i.e., following all control and data dependencies from the changed code) [55,5]. As such, these methods aim at considering conservatively every test requirement affected by the change, resulting in sets with a large number of irrelevant requirements. Nevertheless, applying such an approach to mutation testing is equivalent to mutating the sliced program. This of course inherits all the limitations of program slicing such as scalability and precision [6], it is conservative (results in large number of false positives) and does not account for equivalent mutants located on potentially infected code.

To circumvent the problems of coverage, researchers have proposed the propagation-based techniques [4,57,58,59], which aim at identifying the program paths that are affected by the program changes. They rely on dependence analysis and symbolic execution to form propagation conditions and decide whether changes propagate to a user-defined distance. Although promising, these techniques are complex and inherit the limitations from symbolic execution.

Researchers have also investigated techniques to automatically augment test suites by generating tests that trigger program output differences [54], increase coverage [63] and increase mutation score [62,61]. Along the same lines differential symbolic execution [52], KATCH [45] and Shadow symbolic execution [32] aim at generating tests that exercise the semantic differences between program versions by incrementally searching the program path space from the changed locations and onwards. These methods are somehow complementary to ours as they can be used to create tests that satisfy the commit-relevant test requirements.

Interestingly, the problem of commit-relevant test requirements has not been investigated by the mutation testing literature [50]. Perhaps the closest work to ours is the regression mutation testing by Zhang *et al.* [67] and the

[7] The paper presents a subset of our results. Our data and results are openly accessible on the following Github link: `https://github.com/relevantMutationTesting`

predictive mutation testing by Zhang *et al.* [65] and Mao *et al.* [43]. Regression mutation testing aims at identifying affected mutants in order to incrementally calculate mutation score, while predictive mutation testing aims at estimating the mutation score without mutant execution. Apart from the different focus (we focus on commit-relevant mutants and refined score, while they focus on speeding up test execution and mutation score) and approach details, our fundamental difference is that we statically target killable mutants (both killed and live by the employed test suites) that are relevant to the changed code (we ignore irrelevant code parts and mutants).

Mutation-based test prioritisation has been studied w.r.t the appropriate mutant operator [41], mutant priority [38], mutants in the change [40] or the diversity of the mutants [60]. However, none of them directly use the mutants related to the program behaviour change in the test prioritisation. Our experiments show that most of the relevant mutants are outside the commit change code. Commit-relevant mutants are more purposeful for the test prioritisation in the regression testing.

10 Conclusion

We proposed commit-aware mutation testing, a mutation-based assessment metric capable of measuring the extent to which the program behaviors affected by some committed changes have been tested. We showed that commit-aware mutation testing has a weak correlation with the traditional mutation score and other regression testing approximations (such as the delta on mutation score between the pre- and post- commit versions and mutants located on modified code), indicating that it is a distinct metric. Furthermore, we investigated and concluded that the relevant-mutants set is a distinct mutant set that cannot be found or expressed through proxies in different mutant classes. Our results also showed that traditional mutant selection is non-optimal for evolving and commit-oriented systems as it loses approximately 50%-60% of the commit-relevant mutants when analyzing 5-25 mutants. Moreover, we demonstrated that by focusing attention on commit relevant mutants, over randomly selected ones and the mutants occurring on a modification, one has 30% more chances of revealing commit-introducing faults. Additionally, to provide further evidence of the importance and diversity of commit-relevant mutants' applicability, we demonstrate a potential use case of the commit-relevant mutants and illustrate their application in evaluating regression test-case prioritization techniques. We show that commit-relevant mutants can be used to evaluate test case prioritization techniques.

In future, we plan to study relevant mutants and their occurrence through commit-history. The exploratory study of relevant mutants will shed more light on the properties of this particular category of mutants and their usability. Moreover, we want to study the properties of mutants in combination with commit-changes properties to identify potential correlations that can lead to more autonomous techniques and the development of machine learning models for automatic commit-relevant mutant selection.

References

1. Ammann, P., Delamaro, M.E., Offutt, J.: Establishing theoretical minimal sets of mutants. In: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation, pp. 21–30. IEEE (2014)
2. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press (2008). DOI 10.1017/CBO9780511809163. URL https://doi.org/10.1017/CBO9780511809163
3. Andrews, J.H., Briand, L.C., Labiche, Y., Namin, A.S.: Using mutation analysis for assessing and comparing testing coverage criteria. IEEE Trans. Software Eng. **32**(8), 608–624 (2006). DOI 10.1109/TSE.2006.83. URL https://doi.org/10.1109/TSE.2006.83
4. Apiwattanapong, T., Santelices, R.A., Chittimalli, P.K., Orso, A., Harrold, M.J.: MATRIX: maintenance-oriented testing requirements identifier and examiner. In: Testing: Academia and Industry Conference - Practice And Research Techniques (TAIC PART 2006), 29-31 August 2006, Windsor, United Kingdom, pp. 137–146 (2006). DOI 10.1109/TAIC-PART.2006.18. URL https://doi.org/10.1109/TAIC-PART.2006.18
5. Binkley, D.W.: Semantics guided regression test cost reduction. IEEE Trans. Software Eng. **23**(8), 498–516 (1997). DOI 10.1109/32.624306. URL https://doi.org/10.1109/32.624306
6. Binkley, D.W., Gold, N.E., Harman, M., Islam, S.S., Krinke, J., Yoo, S.: ORBS and the limits of static slicing. In: 15th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2015, Bremen, Germany, September 27-28, 2015, pp. 1–10 (2015). DOI 10.1109/SCAM.2015.7335396. URL https://doi.org/10.1109/SCAM.2015.7335396
7. Böhme, M., Roychoudhury, A.: Corebench: studying complexity of regression errors. In: International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014, pp. 105–115 (2014). DOI 10.1145/2610384.2628058. URL https://doi.org/10.1145/2610384.2628058
8. Böhme, M., d. S. Oliveira, B.C., Roychoudhury, A.: Regression tests to expose change interaction errors. In: Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013, pp. 334–344 (2013). DOI 10.1145/2491411.2491430. URL https://doi.org/10.1145/2491411.2491430
9. Budd, T.A., Angluin, D.: Two Notions of Correctness and Their Relation to Testing. Acta Informatica **18**(1), 31–45 (1982)
10. Cachia, M.A., Micallef, M., Colombo, C.: Towards incremental mutation testing. Electr. Notes Theor. Comput. Sci. **294**, 2–11 (2013). DOI 10.1016/j.entcs.2013.02.012. URL https://doi.org/10.1016/j.entcs.2013.02.012
11. Cadar, C., Dunbar, D., Engler, D.: Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI'08, p. 209–224. USENIX Association, USA (2008)
12. Chekam, T.T., Papadakis, M., Bissyandé, T.F., Traon, Y.L., Sen, K.: Selecting fault revealing mutants. Empirical Software Engineering **25**(1), 434–487 (2020). DOI 10.1007/s10664-019-09778-7. URL https://doi.org/10.1007/s10664-019-09778-7
13. Chekam, T.T., Papadakis, M., Cordy, M., Traon, Y.L.: Killing stubborn mutants with symbolic execution (2020). URL http://arxiv.org/abs/2001.02941
14. Chekam, T.T., Papadakis, M., Traon, Y.L.: Mart: a mutant generation tool for LLVM. In: Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019, pp. 1080–1084 (2019). DOI 10.1145/3338906.3341180. URL https://doi.org/10.1145/3338906.3341180

15. Chekam, T.T., Papadakis, M., Traon, Y.L.: Muteria: An extensible and flexible multi-criteria software testing framework. In: In AST '20: International Conference on Automation of Software Test (AST '20), October 7–8, 2020, Seoul, Republic of Korea. ACM, New York, NY, USA, 4 pages (2020). DOI 10.1145/3387903.3389316. URL `https://doi.org/10.1145/3387903.3389316`
16. Chekam, T.T., Papadakis, M., Traon, Y.L., Harman, M.: An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In: Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017, pp. 597–608 (2017). DOI 10.1109/ICSE.2017.61. URL `https://doi.org/10.1109/ICSE.2017.61`
17. Coles, H., Laurent, T., Henard, C., Papadakis, M., Ventresque, A.: PIT: a practical mutation testing tool for java (demo). In: Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016, pp. 449–452 (2016). DOI 10.1145/2931037.2948707. URL `https://doi.org/10.1145/2931037.2948707`
18. Di Nardo, D., Alshahwan, N., Briand, L., Labiche, Y.: Coverage-based regression test case selection, minimization and prioritization: A case study on an industrial system. Software Testing, Verification and Reliability **25**(4), 371–396 (2015)
19. Do, H., Rothermel, G.: On the use of mutation faults in empirical assessments of test case prioritization techniques. IEEE Transactions on Software Engineering **32**(9), 733–752 (2006)
20. Evans, R.B., Savoia, A.: Differential testing: a new approach to change detection. In: Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007, pp. 549–552 (2007). DOI 10.1145/1287624.1287707. URL `http://doi.acm.org/10.1145/1287624.1287707`
21. Falleri, J., Morandat, F., Blanc, X., Martinez, M., Monperrus, M.: Fine-grained and accurate source code differencing. In: ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014, pp. 313–324 (2014). DOI 10.1145/2642937.2642982. URL `http://doi.acm.org/10.1145/2642937.2642982`
22. Fang, C., Chen, Z., Xu, B.: Comparing logic coverage criteria on test case prioritization. Science China Information Sciences **55**(12), 2826–2840 (2012)
23. Fraser, G., Zeller, A.: Mutation-driven generation of unit tests and oracles. IEEE Trans. Software Eng. **38**(2), 278–292 (2012). DOI 10.1109/TSE.2011.93. URL `https://doi.org/10.1109/TSE.2011.93`
24. Garg, A., Ojdanic, M., Degiovanni, R., Chekam, T.T., Papadakis, M., Traon, Y.L.: Cerebro: Static subsuming mutant selection. IEEE Transactions on Software Engineering DOI 10.1109/TSE.2022.3140510
25. Gheyi, R., Ribeiro, M., Souza, B., Guimarães, M.A., Fernandes, L., d'Amorim, M., Alves, V., Teixeira, L., Fonseca, B.: Identifying method-level mutation subsumption relations using Z3. Inf. Softw. Technol. **132**, 106496 (2021). DOI 10.1016/j.infsof.2020.106496. URL `https://doi.org/10.1016/j.infsof.2020.106496`
26. Hariri, F., Shi, A., Fernando, V., Mahmood, S., Marinov, D.: Comparing mutation testing at the levels of source code and compiler intermediate representation. In: 12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019, pp. 114–124 (2019). DOI 10.1109/ICST.2019.00021. URL `https://doi.org/10.1109/ICST.2019.00021`
27. Henard, C., Papadakis, M., Harman, M., Jia, Y., Le Traon, Y.: Comparing white-box and black-box test prioritization. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE), pp. 523–534 (2016). DOI 10.1145/2884781.2884791
28. Jia, Y., Harman, M.: Higher order mutation testing. Information & Software Technology **51**(10), 1379–1393 (2009). DOI 10.1016/j.infsof.2009.04.016. URL `https://doi.org/10.1016/j.infsof.2009.04.016`
29. Khatibsyarbini, M., Isa, M.A., Jawawi, D.N., Tumeng, R.: Test case prioritization approaches in regression testing: A systematic literature review. Information and Software Technology **93**, 74–93 (2018)

30. Kintis, M., Papadakis, M., Jia, Y., Malevris, N., Traon, Y.L., Harman, M.: Detecting trivial mutant equivalences via compiler optimisations. IEEE Trans. Software Eng. **44**(4), 308–333 (2018). DOI 10.1109/TSE.2017.2684805. URL `https://doi.org/10.1109/TSE.2017.2684805`
31. Kintis, M., Papadakis, M., Malevris, N.: Evaluating mutation testing alternatives: A collateral experiment. In: J. Han, T.D. Thu (eds.) 17th Asia Pacific Software Engineering Conference, APSEC 2010, Sydney, Australia, November 30 - December 3, 2010, pp. 300–309. IEEE Computer Society (2010). DOI 10.1109/APSEC.2010.42. URL `https://doi.org/10.1109/APSEC.2010.42`
32. Kuchta, T., Palikareva, H., Cadar, C.: Shadow symbolic execution for testing software patches. ACM Trans. Softw. Eng. Methodol. **27**(3), 10:1–10:32 (2018). DOI 10.1145/3208952. URL `https://doi.org/10.1145/3208952`
33. Kurtz, B., Ammann, P., Delamaro, M.E., Offutt, J., Deng, L.: Mutant subsumption graphs. In: 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops, pp. 176–185. IEEE (2014)
34. Kurtz, B., Ammann, P., Offutt, J., Delamaro, M.E., Kurtz, M., Gökçe, N.: Analyzing the validity of selective mutation with dominator mutants. In: Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016, pp. 571–582 (2016). DOI 10.1145/2950290.2950322. URL `https://doi.org/10.1145/2950290.2950322`
35. Lam, W., Godefroid, P., Nath, S., Santhiar, A., Thummalapenta, S.: Root causing flaky tests in a large-scale industrial setting. ISSTA 2019, p. 101–111. Association for Computing Machinery, New York, NY, USA (2019). DOI 10.1145/3293882.3330570. URL `https://doi.org/10.1145/3293882.3330570`
36. Laurent, T., Papadakis, M., Kintis, M., Henard, C., Le Traon, Y., Ventresque, A.: Assessing and improving the mutation testing practice of pit. In: 2017 IEEE International Conference on Software Testing, Verification and Validation (ICST), pp. 430–435. IEEE (2017)
37. Leong, C., Singh, A., Papadakis, M., Traon, Y.L., Micco, J.: Assessing transition-based test selection algorithms at google. In: Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019, pp. 101–110 (2019). DOI 10.1109/ICSE-SEIP.2019.00019. URL `https://doi.org/10.1109/ICSE-SEIP.2019.00019`
38. Li, L., Zhou, Y., Yu, Y., Zhao, F., Wu, S., Yang, Z.: An empirical study of mutation-based test case clustering prioritization and reduction technique. ICSEA 2019 p. 13 (2019)
39. Li, N., Praphamontripong, U., Offutt, J.: An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In: 2009 International Conference on Software Testing, Verification, and Validation Workshops, pp. 220–229. IEEE (2009)
40. Lou, Y., Hao, D., Zhang, L.: Mutation-based test-case prioritization in software evolution. In: 2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE), pp. 46–57 (2015). DOI 10.1109/ISSRE.2015.7381798
41. Luo, Q., Moran, K., Poshyvanyk, D., Di Penta, M.: Assessing test case prioritization on real faults and mutants. In: 2018 IEEE international conference on software maintenance and evolution (ICSME), pp. 240–251. IEEE (2018)
42. Ma, W., Laurent, T., Ojdanić, M., Chekam, T.T., Ventresque, A., Papadakis, M.: Commit-aware mutation testing. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 394–405. IEEE (2020)
43. Mao, D., Chen, L., Zhang, L.: An extensive study on cross-project predictive mutation testing. In: 12th IEEE Conference on Software Testing, Validation and Verification, ICST 2019, Xi'an, China, April 22-27, 2019, pp. 160–171 (2019). DOI 10.1109/ICST.2019.00025. URL `https://doi.org/10.1109/ICST.2019.00025`
44. Marcozzi, M., Bardin, S., Kosmatov, N., Papadakis, M., Prevosto, V., Correnson, L.: Time to clean your test objectives. In: M. Chaudron, I. Crnkovic, M. Chechik, M. Harman (eds.) Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, pp. 456–467. ACM (2018). DOI 10.1145/3180155.3180191. URL `https://doi.org/10.1145/3180155.3180191`

45. Marinescu, P.D., Cadar, C.: KATCH: high-coverage testing of software patches. In: Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013, pp. 235–245 (2013). DOI 10.1145/2491411.2491438. URL `https://doi.org/10.1145/2491411.2491438`
46. Offutt, A.J., Rothermel, G., Zapf, C.: An experimental evaluation of selective mutation. In: Proceedings of the 15th International Conference on Software Engineering, Baltimore, Maryland, USA, May 17-21, 1993, pp. 100–107 (1993). URL `http://portal.acm.org/citation.cfm?id=257572.257597`
47. Papadakis, M., Chekam, T.T., Traon, Y.L.: Mutant quality indicators. In: 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops, Västerås, Sweden, April 9-13, 2018, pp. 32–39. IEEE Computer Society (2018). DOI 10.1109/ICSTW.2018.00025. URL `http://doi.ieeecomputersociety.org/10.1109/ICSTW.2018.00025`
48. Papadakis, M., Henard, C., Harman, M., Jia, Y., Traon, Y.L.: Threats to the validity of mutation-based test assessment. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016, pp. 354–365 (2016). DOI 10.1145/2931037.2931040. URL `https://doi.org/10.1145/2931037.2931040`
49. Papadakis, M., Jia, Y., Harman, M., Traon, Y.L.: Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In: A. Bertolino, G. Canfora, S.G. Elbaum (eds.) 37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1, pp. 936–946. IEEE Computer Society (2015). DOI 10.1109/ICSE.2015.103. URL `https://doi.org/10.1109/ICSE.2015.103`
50. Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Traon, Y.L., Harman, M.: Chapter six - mutation testing advances: An analysis and survey. Advances in Computers **112**, 275–378 (2019). DOI 10.1016/bs.adcom.2018.03.015. URL `https://doi.org/10.1016/bs.adcom.2018.03.015`
51. Papadakis, M., Shin, D., Yoo, S., Bae, D.: Are mutation scores correlated with real fault detection?: a large scale empirical study on the relationship between mutants and real faults. In: Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018, pp. 537–548 (2018). DOI 10.1145/3180155.3180183. URL `https://doi.org/10.1145/3180155.3180183`
52. Person, S., Dwyer, M.B., Elbaum, S.G., Pasareanu, C.S.: Differential symbolic execution. In: Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008, pp. 226–237 (2008). DOI 10.1145/1453101.1453131. URL `https://doi.org/10.1145/1453101.1453131`
53. Petrovic, G., Ivankovic, M.: State of mutation testing at google. In: Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2018, Gothenburg, Sweden, May 27 - June 03, 2018, pp. 163–171 (2018). DOI 10.1145/3183519.3183521. URL `https://doi.org/10.1145/3183519.3183521`
54. Qi, D., Roychoudhury, A., Liang, Z.: Test generation to expose changes in evolving programs. In: ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010, pp. 397–406 (2010). DOI 10.1145/1858996.1859083. URL `https://doi.org/10.1145/1858996.1859083`
55. Rothermel, G., Harrold, M.J.: Selecting tests and identifying test coverage requirements for modified software. In: Proceedings of the 1994 International Symposium on Software Testing and Analysis, ISSTA 1994, Seattle, WA, USA, August 17-19, 1994, pp. 169–184 (1994). DOI 10.1145/186258.187171. URL `https://doi.org/10.1145/186258.187171`
56. Rothermel, G., Untch, R.H., Chu, C., Harrold, M.J.: Test case prioritization: An empirical study. In: Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360), pp. 179–188. IEEE (1999)
57. Santelices, R.A., Chittimalli, P.K., Apiwattanapong, T., Orso, A., Harrold, M.J.: Test-suite augmentation for evolving software. In: 23rd IEEE/ACM International Conference

on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy, pp. 218–227 (2008). DOI 10.1109/ASE.2008.32. URL `https://doi.org/10.1109/ASE.2008.32`

58. Santelices, R.A., Harrold, M.J.: Exploiting program dependencies for scalable multiple-path symbolic execution. In: Proceedings of the Nineteenth International Symposium on Software Testing and Analysis, ISSTA 2010, Trento, Italy, July 12-16, 2010, pp. 195–206 (2010). DOI 10.1145/1831708.1831733. URL `https://doi.org/10.1145/1831708.1831733`
59. Santelices, R.A., Harrold, M.J.: Applying aggressive propagation-based strategies for testing changes. In: Fourth IEEE International Conference on Software Testing, Verification and Validation, ICST 2011, Berlin, Germany, March 21-25, 2011, pp. 11–20 (2011). DOI 10.1109/ICST.2011.46. URL `https://doi.org/10.1109/ICST.2011.46`
60. Shin, D., Yoo, S., Papadakis, M., Bae, D.H.: Empirical evaluation of mutation-based test case prioritization techniques. Software Testing, Verification and Reliability **29**(1-2), e1695 (2019)
61. Smith, B.H., Williams, L.: On guiding the augmentation of an automated test suite via mutation analysis. Empirical Software Engineering **14**(3), 341–369 (2009). DOI 10.1007/s10664-008-9083-7. URL `https://doi.org/10.1007/s10664-008-9083-7`
62. Smith, B.H., Williams, L.: Should software testers use mutation analysis to augment a test set? Journal of Systems and Software **82**(11), 1819–1832 (2009). DOI 10.1016/j.jss.2009.06.031. URL `https://doi.org/10.1016/j.jss.2009.06.031`
63. Xu, Z., Kim, Y., Kim, M., Rothermel, G., Cohen, M.B.: Directed test suite augmentation: techniques and tradeoffs. In: Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2010, Santa Fe, NM, USA, November 7-11, 2010, pp. 257–266 (2010). DOI 10.1145/1882291.1882330. URL `https://doi.org/10.1145/1882291.1882330`
64. Yoo, S., Harman, M.: Regression testing minimization, selection and prioritization: a survey. Softw. Test., Verif. Reliab. **22**(2), 67–120 (2012). DOI 10.1002/stv.430. URL `https://doi.org/10.1002/stv.430`
65. Zhang, J., Zhang, L., Harman, M., Hao, D., Jia, Y., Zhang, L.: Predictive mutation testing. IEEE Trans. Software Eng. **45**(9), 898–918 (2019). DOI 10.1109/TSE.2018.2809496. URL `https://doi.org/10.1109/TSE.2018.2809496`
66. Zhang, L., Hao, D., Zhang, L., Rothermel, G., Mei, H.: Bridging the gap between the total and additional test-case prioritization strategies. In: 2013 35th International Conference on Software Engineering (ICSE), pp. 192–201. IEEE (2013)
67. Zhang, L., Marinov, D., Zhang, L., Khurshid, S.: Regression mutation testing. In: International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012, pp. 331–341 (2012). DOI 10.1145/2338965.2336793