Commit-Aware Mutation Testing

Wei Ma*, Thomas Laurent †, Miloš Ojdanić*, Thierry Titcheu Chekam*, Anthony Ventresque†, and Mike Papadakis*
*SnT, University of Luxembourg, Luxembourg
†Lero School of Computer Science, University College Dublin
*{firstname.surname}@uni.lu, †thomas.laurent@ucdconnect.ie, anthony.ventresque@ucd.ie

Abstract—**In Continuous Integration, developers want to know how well they have tested their changes. Unfortunately, in these cases, the use of mutation testing is suboptimal since mutants affect the entire set of program behaviours and not the changed ones. Thus, the extent to which mutation testing can be used to test committed changes is questionable. To deal with this issue, we define commit-relevant mutants; a set of mutants that affect the changed program behaviours and represent the commit-relevant test requirements. We identify such mutants in a controlled way, and check their relationship with traditional mutation score (score based on the entire set of mutants or on the mutants located on the commits). We conduct experiments in both C and Java, using 83 commits, 2,253,610 mutants from 25 projects. Our findings reveal that there is a relatively weak correlation (Kendall/Pearson 0.15-0.4) between the sought (commit-relevant) and traditional mutation scores, indicating the need for a commit-aware test assessment metric. Our analysis also shows that traditional mutation is far from the envisioned case as it loses approximately 50%-60% of the commit-relevant mutants when analysing 5-25 mutants. More importantly, our results demonstrate that traditional mutation has approximately 30% lower chances of revealing commit-introducing faults than commit-aware mutation testing.**

I. Introduction

Modern software development involves the continuous submission and integration of code modifications from many developers into a common codebase [1]. This continuous development is performed by automatic procedures that build and test the software products. Automated testing is used to establish confidence that the committed code behaves correctly, while at the same time it does not break any of the previously developed program functionalities.

When developers commit their code, they are interested in testing the delta of behaviours between their pre- and post-commit versions in order to discover issues and side effects caused by their changes. Thus, developers are interested in knowing how well they have tested the program behaviours affected by their changes. To this end, many studies suggest using mutation testing (or other test adequacy criteria) to drive test generation, or to assess test thoroughness on the evolving software [2], [3].

Mutation testing has long been established as one of the strongest test criteria [4]. 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 test cases [5] or to measure test suites' thoroughness [6].

Previous research has shown that mutation testing leads to fault revelation [7], [8] and can be used for test assessment as it effectively quantifies the test suites' strengths [6]. Unfortunately, traditional mutation testing aim at testing the entire codebase, rather than specific program changes/commits as would naturally be requested by developers.

There are many studies aiming at making the mutation score metric accurate either by using specific mutant types [9], or by detecting equivalent mutants [10], [11], i.e., mutants that cannot be killed by any test case because they are semantically equivalent to the original program, or by eliminating redundant mutants [12], [13], i.e., mutants that are killed "collaterally" whenever other mutants are killed (subsumed by the subsuming mutants). Yet, little research has focused on measuring the effectiveness of test suites with respect to particular program changes or commits.

To form a commit-aware mutation criterion, it is necessary to identify mutants capturing the altered program behaviours, i.e., mutants interacting with the changed program behaviours, representing the sought commit-relevant test requirements. These mutants can then be used to judge whether test suites are adequate and, if not, to provide guidance in improving test suites (by creating tests that kill commit-relevant mutants).

One may assume that, since mutation score reflects test thoroughness (of the whole system, component or class under test), it also reflects, or at least the score delta between versions reflects, the extent to which changes are tested. Someone else may consider that the changed program parts can be tested by mutating only the modified code, assuming that mutant locations reflect their utility and relevance.

These assumptions may appear intuitive but unfortunately they do not hold. This is because of the large numbers of irrelevant (to the committed changes) mutants and the many relevant ones that are spread on the entire codebase. Since these mutants are unknown to mutation testers, they hinder their ability to distinguish between relevant and irrelevant mutant kills. Mutating only the modified code parts yields better results, but still, it is 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 [14], [15]. This is demonstrated by our results that show the majority of the altered program behaviours to be captured by mutants located on unmodified code parts.

In our analysis we also considered the potential gains and losses of either using the entire set of mutants or those mutants that are located on the committed code. Obviously, by killing all the mutants, one achieves killing all the commit-relevant ones. However, this comes with the cost of analysing more mutants, and generating more test cases than needed. Perhaps more importantly, the killing of mutants irrelevant to the commit inflates the mutation score, hindering its ability to reflect test thoroughness w.r.t. to committed code. Similarly, by killing all the mutants located on the committed code, one fails to kill a significant number of commit-relevant mutants, loosing significant test effectiveness.

Interestingly, our results reveal that there is a relatively weak correlation between the sought commit-aware and traditional mutation scores, indicating the need for a commit-relevant test assessment metric. Our analysis also shows that when using mutants for test suite improvement [16] (by adding tests that kill mutants), traditional mutant selection is very far from the envisioned case, as it loses approximately 50%-60% of the commit-relevant mutants (when analysing 5-25 mutants). Perhaps more importantly, our results demonstrate that commit-relevant mutants have 30% more chances to reveal faults (real faults) than traditional mutation when analysing the same number of mutants (putting approximately the same amount of effort).

Overall, our contribution is the definition of the commit-relevant mutants and the envisioned commit-relevant mutation-based test assessment. We motivate this by providing evidence that mutation testing performed with the entire set of mutants or with the mutants located on the committed code is insufficient to assess test thoroughness or to provide cost-effective guidance to adequately test particular program changes.

Taken together, our key contributions can be summarised by the following points:

- We define the 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 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 sought 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 (wrt real commit-introduced faults) than the other strategies when analysing the same number of mutants.

II. Mutation Testing

A. Test Criteria and Mutation Testing

Test criteria are metrics quantifying the extent to which systems are tested [7]. 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 [17] or test selection [18]. 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 [12].

Mutation analysis is a test criterion [19] 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 differencitate 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.

B. Equivalent and Duplicated 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 [4], as identifying them is an undecidable problem [20].

Interestingly, many killable mutants are equivalent to others, introducing another problem, skew in the Mutation Score. Kintis *et al.* [10] have shown this to be problematic and suggest to getting rid of these “duplicated” mutants (mutants equivalent to others).

C. Regression Mutation Testing

Applying mutation during regression testing has long been proposed. In particular, Cachia *et al.* [21] proposed applying change-based mutation testing by considering only the mutants located on the altered code. Zhang *et al.* [2] 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 [22], 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 [3] 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.

III. 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.

Precisely, the regression changes interact with a mutant when the program version that includes both the regression changes and the mutant behaves differently from:

1) the version that includes only the mutant (mutant in the pre-commit version).
2) the version that includes only the regression changes (post-commit version).

This situation is depicted in Figure 1.

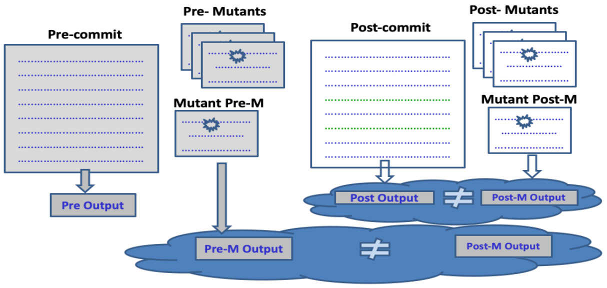


Fig. 1. A mutant is relevant if it impacts the behaviour of the committed code and the committed code impacts the behaviour of the mutant.

A. Demonstrating Example

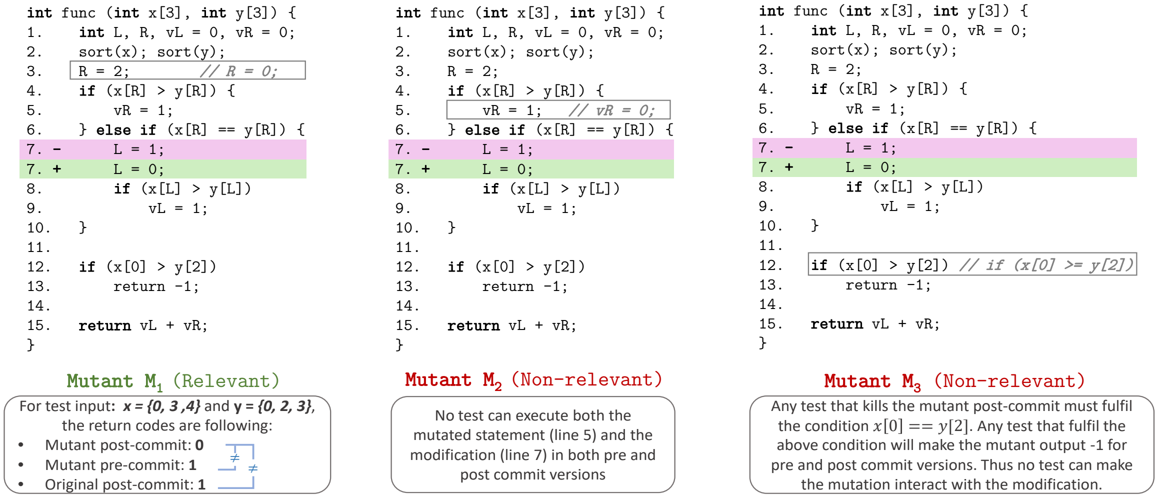
Figure 2 illustrates the concept of relevant mutants. The example function takes 2 arguments (integer arrays *x* and *y* of size 3), sorts them, makes some computations, and outputs an integer. The commit modification alters the statement at line *7* by changing the value assigned to the variable *L* 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.

The sub-figure on the left side shows mutant M_1. M_1 is characterized 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.



```
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.

IV. Experimental Setup

A. 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 [16]. 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 [13], [23]. 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 [8], [13] 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 compares with the 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.

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

B. Analysis Procedure

We performed mutation testing on the selected subjects using all the mutation operators supported by Mart [24] and Pitest [22] (the mutation testing tools we use). For the C programs, we then discarded all the trivially equivalent mutants (including the duplicated ones), using the TCE method [10] and applied our analysis on the resulting sets of mutants.

Identifying relevant mutants requires excessive manual analysis, thus we approximate them based on test suites (this is a typical experimental procedure [25], [13], [4]). 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 and Papadakis et al. [13], [4].

Using the test pools, we execute all the mutants (on both pre- and post-commit versions) and construct the mutation matrix that records the mutants killed by each test case of the pool. We also record the test execution output of each test on each mutants. For C programs, this 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 the test execution outputs and the mutant matrices, we approximate the relevant mutant set, from the post-commit mutants, 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.

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.

We have three mutant sets: the post-commit, relevant and modification mutant sets. 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.

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*), which instances are the corresponding mutation scores for each test set.

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 $RelevantMuts$ ;
```

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 [13], [4] 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.

C. 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 1% 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 1% indicates that the two metrics are statistically different. We use paired and two-tailed U test, to account for the different commits and programs.

D. Program Versions Used

To answer RQs 1-3 we used the C programs of GNU Coreutils[1], used in many existing studies [26], [27], [28]. 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, we filtered out commits that are very large (commits whose modification has an edit actions of more than 5 according to GumTree [29]). 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 [30] 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 [26]. 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 I
C Test Subjects

Benchmark	#Programs	#Commits	# Mutants	#Test cases
CoREBench [30]	6	13	154,396	8,828
Benchmark-1	13	34	338,390	11,866

TABLE II
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

Table I summarizes the informations 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 II summarises information about the commits used from each project.

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

[1] https://www.gnu.org/software/coreutils/

[2] Measured with cloc (http://cloc.sourceforge.net/)

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

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

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

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 [26], 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 [26].

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 and the commit diff. We first generate 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, and use this mapping 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).

F. 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 [28], with a robust timeout of 2 hours, to perform a form of differential testing [31] called shadow symbolic execution [26], 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 [27], 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 [24], 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) [10], [32] 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 developper test suites available. We perform mutation analysis using Pitest[22], 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 [33] and [34].

V. Results

A. 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.

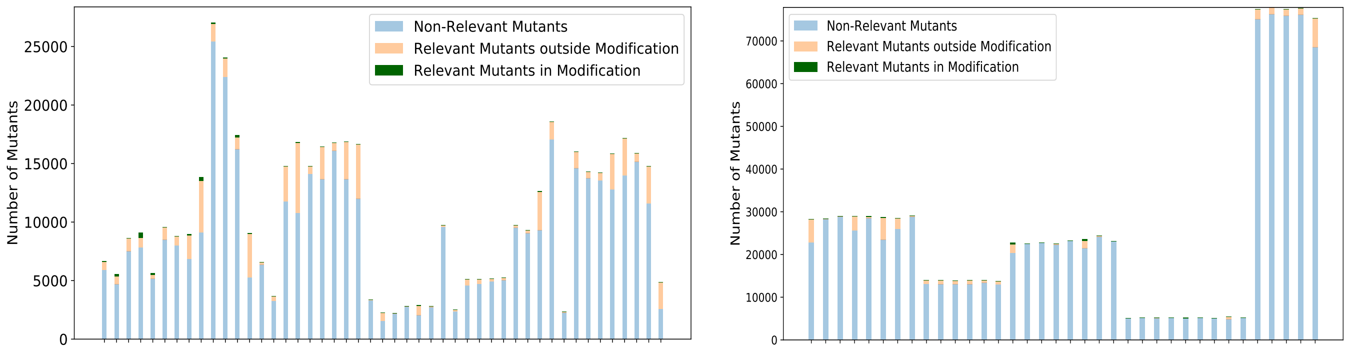
Interestingly, the presence of so many "irrelevant" mutants, can have major consequences when performing mutation testing. Such consequences are a distorting effect on the accuracy of the mutation score, and a waste of resources when executing and trying to kill non-relevant to the commit mutants. We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

B. 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.

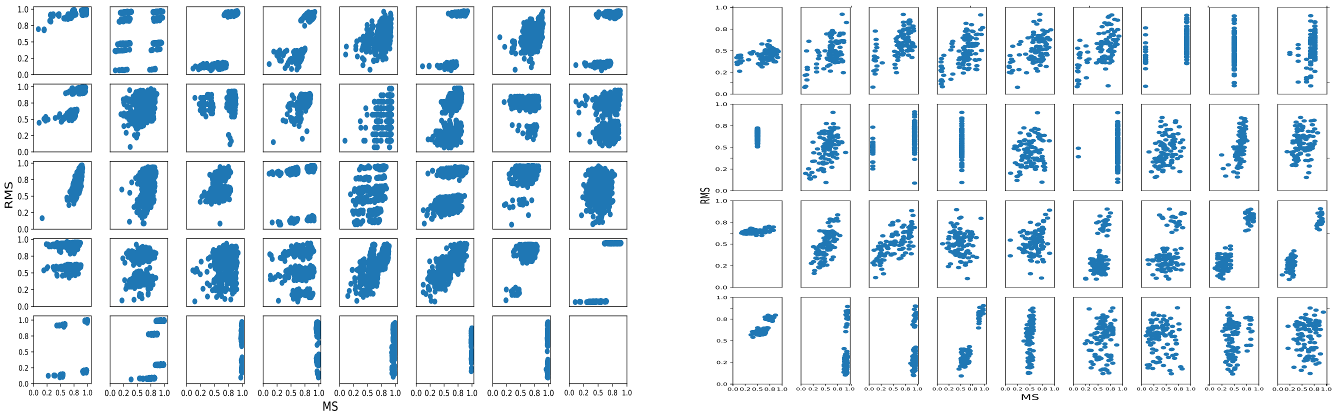
Finding a strong correlation would suggest that the two metrics have similar behaviours (an increase or decrease of one implies a relatively similar increase or decrease of the other). Figure 5 displays the results for the two correlation coefficients that have statistically significant values for randomly 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.

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



(a) C PROGRAMS (b) JAVA PROGRAMS

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

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 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.

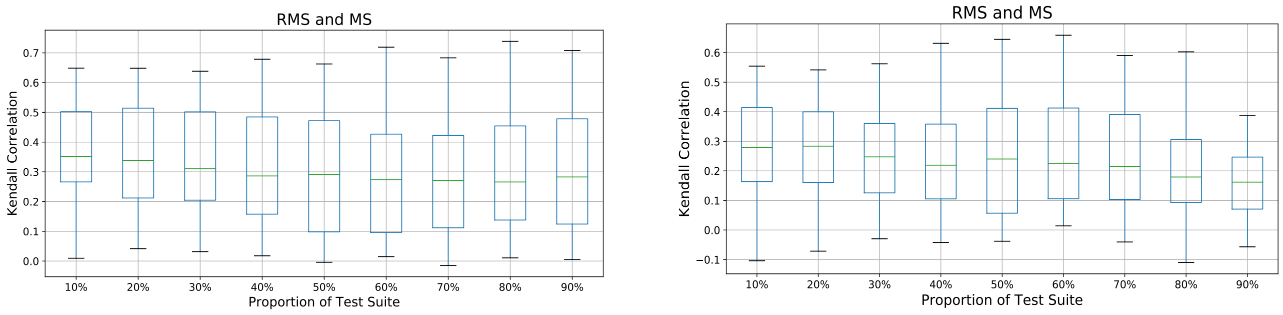
C. RQ3: Test Selection

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 [16]. 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.

TABLE III
$\hat{A}_{12}$. 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

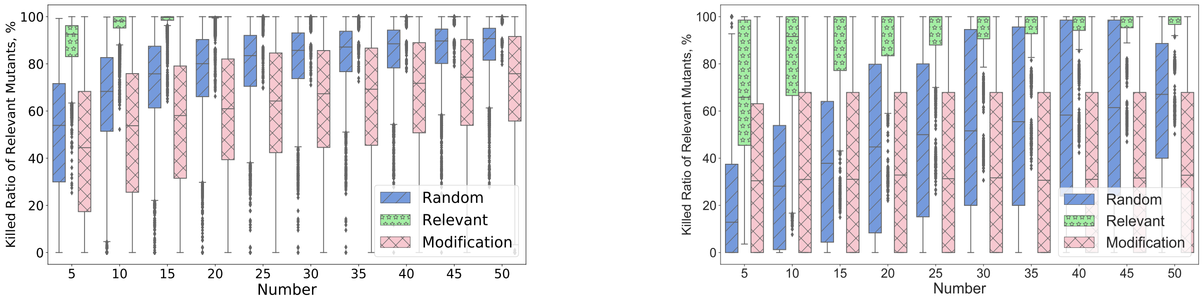
Results are recorded in Figure 6 for the first 1-50 mutants to be analysed by the tester. We observe a large divergence (approximately 50%-60%) between the random, commit-based and relevant mutants. This difference is statistically significant and with large effect size (Effect Size values are recorded on Table III). Taking together the weak correlations we found in the previous section with these results, we conclude that traditional mutation testing is suboptimal 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.



(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 IV

$\hat{A}_{12}$. 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

D. 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 IV). 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.

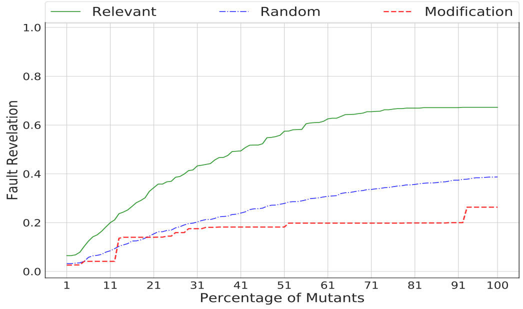


Fig. 7. Fault revelation of mutation-based testing with random (traditional mutation) and relevant mutants.

VI. Threats to Validity

External validity: We selected commits that do not modify test contracts. Such commits are common in industrial CI pipelines [35] 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 [**?**]). In addition, we used an established research benchmark (CoREBench [30]) 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 [33], in both C and Java.

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 [3] 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 [10] to remove all trivially equivalent mutants before conducting any experiment and KLEE (including a mutation-based test generation approach [27]). 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 [36] 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 [4]. Additionally, to enable reproducibility and replication we make our tools and data publicly available[7].

VII. 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) [37], [38]. 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 [39], 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 [40], [15], [41], [42], 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 [43], increase coverage [44] and increase mutation score [45], [46]. Along the same lines differential symbolic execution [47], KATCH [48] and Shadow symbolic execution [26] 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 [4]. Perhaps the closest work to ours is the regression mutation testing by Zhang et al. [2] and the predictive mutation testing by Zhang et al. and Mao et al. [36], [49]. 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).

VIII. Conclusion

We proposed commit-aware mutation testing, a mutation-based assessment metric capable of measuring the extent to which the program behaviours 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. Our results also showed that traditional mutant selection is non-optimal as it loses approximately 50%-60% of the commit-relevant mutants when analysing 5-25 mutants and has 30% less chances of revealing commit-introducing faults.

Acknowledgement

This work is supported by the Luxembourg National Research Funds (FNR) through the CORE project grant C17/IS/11686509/CODEMATES and by the Science Foundation Ireland grant 13/RC/2094. T. Laurent is supported by an Irish Research Council grant (GOIPG/2017/1829).

[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

References

[1] M. Fowler, "Continuous integration," https://martinfowler.com/articles/continuousIntegration.html, online; accessed 10 February 2020.

[2] L. Zhang, D. Marinov, L. Zhang, and S. Khurshid, "Regression mutation testing," in *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*, 2012, pp. 331–341.

[3] G. Petrovic and M. Ivankovic, "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*, 2018, pp. 163–171. [Online]. Available: https://doi.org/10.1145/3183519.3183521

[4] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman, "Chapter six - mutation testing advances: An analysis and survey," *Advances in Computers*, vol. 112, pp. 275–378, 2019. [Online]. Available: https://doi.org/10.1016/bs.adcom.2018.03.015

[5] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, vol. 11, no. 4, pp. 34–41, 1978. [Online]. Available: https://doi.org/10.1109/C-M.1978.218136

[6] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin, "Using mutation analysis for assessing and comparing testing coverage criteria," *IEEE Trans. Software Eng.*, vol. 32, no. 8, pp. 608–624, 2006. [Online]. Available: https://doi.org/10.1109/TSE.2006.83

[7] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, 2008. [Online]. Available: https://doi.org/10.1017/CBO9780511809163

[8] T. T. Chekam, M. Papadakis, Y. L. Traon, and M. Harman, "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*, 2017, pp. 597–608. [Online]. Available: https://doi.org/10.1109/ICSE.2017.61

[9] A. J. Offutt, G. Rothermel, and C. Zapf, "An experimental evaluation of selective mutation," in *Proceedings of the 15th International Conference on Software Engineering, Baltimore, Maryland, USA, May 17-21, 1993*, 1993, pp. 100–107. [Online]. Available: http://portal.acm.org/citation.cfm?id=257572.257597

[10] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. L. Traon, and M. Harman, "Detecting trivial mutant equivalences via compiler optimisations," *IEEE Trans. Software Eng.*, vol. 44, no. 4, pp. 308–333, 2018. [Online]. Available: https://doi.org/10.1109/TSE.2017.2684805

[11] M. Marcozzi, S. Bardin, N. Kosmatov, M. Papadakis, V. Prevosto, and L. Correnson, "Time to clean your test objectives," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 456–467. [Online]. Available: https://doi.org/10.1145/3180155.3180191

[12] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. L. Traon, "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*, 2016, pp. 354–365. [Online]. Available: https://doi.org/10.1145/2931037.2931040

[13] B. Kurtz, P. Ammann, J. Offutt, M. E. Delamaro, M. Kurtz, and N. Gökçe, "Analyzing the validity of selective mutation with dominator mutants," in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, 2016, pp. 571–582. [Online]. Available: https://doi.org/10.1145/2950290.2950322

[14] M. Böhme, B. C. d. S. Oliveira, and A. Roychoudhury, "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*, 2013, pp. 334–344. [Online]. Available: https://doi.org/10.1145/2491411.2491430

[15] R. A. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold, "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*, 2008, pp. 218–227. [Online]. Available: https://doi.org/10.1109/ASE.2008.32

[16] M. Papadakis, D. Shin, S. Yoo, and D. Bae, "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*, 2018, pp. 537–548. [Online]. Available: https://doi.org/10.1145/3180155.3180183

[17] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *IEEE Trans. Software Eng.*, vol. 38, no. 2, pp. 278–292, 2012. [Online]. Available: https://doi.org/10.1109/TSE.2011.93

[18] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Softw. Test., Verif. Reliab.*, vol. 22, no. 2, pp. 67–120, 2012. [Online]. Available: https://doi.org/10.1002/stv.430

[19] N. Li, U. Praphamontripong, and J. Offutt, "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*. IEEE, 2009, pp. 220–229.

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

[21] M. A. Cachia, M. Micallef, and C. Colombo, "Towards incremental mutation testing," *Electr. Notes Theor. Comput. Sci.*, vol. 294, pp. 2–11, 2013. [Online]. Available: https://doi.org/10.1016/j.entcs.2013.02.012

[22] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "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*, 2016, pp. 449–452. [Online]. Available: https://doi.org/10.1145/2931037.2948707

[23] T. T. Chekam, M. Papadakis, T. F. Bissyandé, Y. L. Traon, and K. Sen, "Selecting fault revealing mutants," *Empirical Software Engineering*, vol. 25, no. 1, pp. 434–487, 2020. [Online]. Available: https://doi.org/10.1007/s10664-019-09778-7

[24] T. T. Chekam, M. Papadakis, and Y. L. Traon, "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*, 2019, pp. 1080–1084. [Online]. Available: https://doi.org/10.1145/3338906.3341180

[25] P. Ammann, M. E. Delamaro, and J. Offutt, "Establishing theoretical minimal sets of mutants," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 2014.

[26] T. Kuchta, H. Palikareva, and C. Cadar, "Shadow symbolic execution for testing software patches," *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 3, pp. 10:1–10:32, 2018. [Online]. Available: https://doi.org/10.1145/3208952

[27] T. T. Chekam, M. Papadakis, M. Cordy, and Y. L. Traon, "Killing stubborn mutants with symbolic execution," 2020. [Online]. Available: http://arxiv.org/abs/2001.02941

[28] C. Cadar, D. Dunbar, and D. Engler, "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*, ser. OSDI08. USA: USENIX Association, 2008, p. 209224.

[29] J. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, 2014, pp. 313–324. [Online]. Available: http://doi.acm.org/10.1145/2642937.2642982

[30] M. Böhme and A. Roychoudhury, "Corebench: studying complexity of regression errors," in *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, 2014, pp. 105–115. [Online]. Available: https://doi.org/10.1145/2610384.2628058

[31] R. B. Evans and A. Savoia, "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*, 2007, pp. 549–552. [Online]. Available: http://doi.acm.org/10.1145/1287624.1287707

[32] F. Hariri, A. Shi, V. Fernando, S. Mahmood, and D. Marinov, "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*, 2019, pp. 114–124. [Online]. Available: https://doi.org/10.1109/ICST.2019.00021

[33] T. Laurent, M. Papadakis, M. Kintis, C. Henard, Y. Le Traon, and A. Ventresque, "Assessing and improving the mutation testing practice of pit," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2017, pp. 430–435.

[34] H. Coles, “Pitest mutators,” http://pitest.org/quickstart/mutators/, online; accessed 25 May 2020.

[35] C. Leong, A. Singh, M. Papadakis, Y. L. Traon, and J. Micco, “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*, 2019, pp. 101–110. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP.2019.00019

[36] J. Zhang, L. Zhang, M. Harman, D. Hao, Y. Jia, and L. Zhang, “Predictive mutation testing,” *IEEE Trans. Software Eng.*, vol. 45, no. 9, pp. 898–918, 2019. [Online]. Available: https://doi.org/10.1109/TSE.2018.2809496

[37] G. Rothermel and M. J. Harrold, “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*, 1994, pp. 169–184. [Online]. Available: https://doi.org/10.1145/186258.187171

[38] D. W. Binkley, “Semantics guided regression test cost reduction,” *IEEE Trans. Software Eng.*, vol. 23, no. 8, pp. 498–516, 1997. [Online]. Available: https://doi.org/10.1109/32.624306

[39] D. W. Binkley, N. E. Gold, M. Harman, S. S. Islam, J. Krinke, and S. Yoo, “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*, 2015, pp. 1–10. [Online]. Available: https://doi.org/10.1109/SCAM.2015.7335396

[40] T. Apiwattanapong, R. A. Santelices, P. K. Chittimalli, A. Orso, and M. J. Harrold, “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*, 2006, pp. 137–146. [Online]. Available: https://doi.org/10.1109/TAIC-PART.2006.18

[41] R. A. Santelices and M. J. Harrold, “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*, 2010, pp. 195–206. [Online]. Available: https://doi.org/10.1145/1831708.1831733

[42] ——, “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*, 2011, pp. 11–20. [Online]. Available: https://doi.org/10.1109/ICST.2011.46

[43] D. Qi, A. Roychoudhury, and Z. Liang, “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*, 2010, pp. 397–406. [Online]. Available: https://doi.org/10.1145/1858996.1859083

[44] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen, “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*, 2010, pp. 257–266. [Online]. Available: https://doi.org/10.1145/1882291.1882330

[45] B. H. Smith and L. Williams, “Should software testers use mutation analysis to augment a test set?” *Journal of Systems and Software*, vol. 82, no. 11, pp. 1819–1832, 2009. [Online]. Available: https://doi.org/10.1016/j.jss.2009.06.031

[46] ——, “On guiding the augmentation of an automated test suite via mutation analysis,” *Empirical Software Engineering*, vol. 14, no. 3, pp. 341–369, 2009. [Online]. Available: https://doi.org/10.1007/s10664-008-9083-7

[47] S. Person, M. B. Dwyer, S. G. Elbaum, and C. S. Pasareanu, “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*, 2008, pp. 226–237. [Online]. Available: https://doi.org/10.1145/1453101.1453131

[48] P. D. Marinescu and C. Cadar, “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*, 2013, pp. 235–245. [Online]. Available: https://doi.org/10.1145/2491411.2491438

[49] D. Mao, L. Chen, and L. Zhang, “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*, 2019, pp. 160–171. [Online]. Available: https://doi.org/10.1109/ICST.2019.00025