MuDelta: Delta-Oriented Mutation Testing at Commit Time

Wei Ma*, Thierry Titcheu Chekam*, Mike Papadakis* and Mark Harman†
*SnT, University of Luxembourg, Luxembourg
†Facebook and University College London, UK
*{firstname.surname}@uni.lu, †mark.harman@ucl.ac.uk

***Abstract*—To effectively test program changes using mutation testing, one needs to use mutants that are relevant to the altered program behaviours. We introduce *MuDelta*, an approach that identifies commit-relevant mutants; mutants that affect and are affected by the changed program behaviours. Our approach uses machine learning applied on a combined scheme of graph and vector-based representations of static code features. Our results, from 50 commits in 21 Coreutils programs, demonstrate a strong prediction ability of our approach; yielding 0.80 (ROC) and 0.50 (PR-Curve) AUC values with 0.63 and 0.32 precision and recall values. These predictions are significantly higher than random guesses, 0.20 (PR-Curve) AUC, 0.21 and 0.21 precision and recall, and subsequently lead to strong relevant tests that kill 45% more relevant mutants than randomly sampled mutants (either sampled from those residing on the changed component(s) or from the changed lines). Our results also show that *MuDelta* selects mutants with 27% higher fault revealing ability in fault introducing commits. Taken together, our results corroborate the conclusion that commit-based mutation testing is suitable and promising for evolving software.**

***Index Terms*—mutation testing, commit-relevant mutants, continuous integration, regression testing, machine learning**

I. Introduction

Mutation testing has been shown to be one of the strongest fault-revealing software test adequacy criteria available to software testers [1]. Nevertheless, although mutation testing has been widely studied for over four decades in the scientific literature, the formulation that underpins it has remained largely unchanged since its inception in the 1970s [2], [3].

In this unchanged formulation, a program p is tested by a test suite, T, the adequacy of which is measured in terms of its ability to distinguish executions of p and a set of mutants M. Each mutant in M is a version of p into which a fault has been deliberately inserted, in order to simulate potential real faults, thereby assessing the ability of the test suite T to detect such faults.

The problem with this formulation is that it has not kept pace with recent software engineering practices. Most notably, the assumption of a fixed program p, set of mutants M, and test suite T, is unrealistic; modern software systems undergo regular change, typically in continuous integration environments [4]–[6]. In order to render mutation testing applicable to practising software engineers, a fundamentally new approach to finding suitable mutants is required in which p, T, and M are each continually evolving.

Specifically, we need a mutation testing formulation in which mutants can be found, on the fly, based on their relevance to specific changes to the system under consideration. In this 'evolving mutation testing' approach, both the set of mutants M and the tests that distinguished their behaviours T, are each able to change with each new commit. Such a mutation testing formulation is better suited to industrial practice, e.g., at Google [7], since mutation testing can be applied at commit time, to each code change as it is submitted, thereby keeping pace with the changes to p. More importantly, such an approach will focus the test effort deployed at commit time specifically to the changes in the commit, rather than wasting test effort on re-testing old code.

In order to apply mutation testing on the fly in this manner, we need a fast lightweight approach to determine a priority ordering on a given set of mutants, where priority is determined by the relevance of a mutant to the change in hand. This paper introduces a machine learning-based approach to tackle this problem using a combined scheme of graph and vector-based representations of simple code features that aim at capturing the information (control and data) flow and interactions between mutants and committed code changes. We train the learner on a set of mutants from historical code changes that are labeled with respect to given test suites. The machine learner is subsequently used to predict the priority ordering of the set of mutants to identify those most likely to be relevant to a given change.

This way, once the learner has been trained, it can be used to quickly predict the priority order for the set of mutants in terms of their relevance to unseen changes, as they are submitted into the continuous integration system for review. This allows the tester (and/or some automated test design technology) to focus on those mutants that are most likely to yield tests that are fault revealing for the change in hand.

We implemented our approach in a system called *MuDelta*, and evaluated it on a set of 50 commits from Coreutils wrt a) prediction ability, b) ability to lead to relevant tests (tests killing commit-relevant mutants) and c) ability to reveal faults in fault introducing commits. Our results indicate s strong prediction ability; *MuDelta* yields 0.80 ROC-AUC value, 0.42 F1-score, 0.63 precision and 0.32 recall, while random guesses yield 0.20 F1-score, 0.21 precision and 0.21 recall. Killing the predicted mutants results in killing 45% more relevant mutants than random mutant sampling baselines.

Perhaps more importantly, our results show that our approach leads to mutants with 27% higher fault revealing ability in fault introducing commits. Taken together, our results corroborate the findings that *MuDelta* enables effective delta-relevant mutation testing, i.e., mutation testing targeting the specific code changes of the software system under test.

Our study also reveals some surprising findings, additional results and discussion can be found in our project page[1]. For example, one might believe that mutants that reside in the changed code would be adequate in testing it. However, our empirical findings contradict this natural, but incorrect, assumption. This surprising finding highlights the importance of finding mutants in the unchanged part of the program. This unchanged code that forms a contextual environment into which changes deployed. Such δ-relevant mutants in the context C, for some change, δ, tend to focus on (and reveal issues with) interactions between the change, δ, and the context C into which it is deployed. Developers are less likely to notice these since they are more likely to be familiar with their changes than the existing unchanged code. Such bugs may also be more subtle as they involve unforeseen interactions between parts of the system.

In summary, our primary contributions are:

- The empirical evidence that mutant relevance (to particular program changes) can be captured by simple static source code metrics.
- A machine learning approach, called *MuDelta*, that learns to rank mutants wrt to their utility and relevance to specific code changes.
- Empirical evidence suggesting that *MuDelta* outperforms the traditionally random mutant selection/prioritization method by revealing 45% more relevant mutants, and achieving 27% higher probability to reveal faults in these changes.

II. Context

A. Mutation testing

Mutation testing [2] measures the fault revealing potential of test cases by checking the extend to which artificially seeded faults, called mutants, are triggered. When a behaviour difference between the original program and a mutant is detected, the mutant is considered to be "killed", otherwise the mutant is considered to be "live". The point here is that mutant killing shows an execution failure that was covered, triggered by the altered code and propagated to the observable program output, signifying the potential of the test. The faults are seeded in the code under analysis by making simple syntactic transformations, e.g., replacing the instance of an operator with another one, `if (a < b)` into `if (a ≤ b)`, and they represent the test requirements. This means that the ratio of mutants killed, called Mutation Score (MS), represents the test thoroughness metric.

[1] https://rml464.github.io/mutantlearning/

Injecting faults by altering the syntax of the program may result in semantically equivalent program versions, i.e., versions that behave the same way for all possible inputs. These equivalent versions need to be removed and not taken into consideration, as even a perfect test suite cannot kill them. Unfortunately, equivalent mutants form one of the known problems of mutation testing [8].

Interestingly, many killable mutants are equivalent to others, introducing an additional problem, skew in the Mutation Score [9], [10]. The problem though, is more severe since not all mutants are equally important; many mutants are killed collaterally, and thus, they do not contribute to the testing process [11], [12]. Unfortunately, these collateral kills inflate the mutation score measurement and may lead to wrong conclusions [10]. Therefore, the recent mutation testing literature [8], [10] suggest using the so-called subsuming mutants (computing the subsuming mutation score [12], [13]) when evaluating test effectiveness.

B. Change-aware regression testing

Testing program regressions require test suites to exercise the adequacy of testing wrt to the program changes. In case the used test suites are insufficient, guidance should be given in order to help developers create test cases that specifically target the behaviour deviations introduced by the regressions.

One potential solution to this problem may be based on coverage; one can aim at testing the altered parts of the programs using coverage information. However, the strengths of coverage are known to be limited [1], [14]. Moreover, the most severe regression issues are due to unforeseen interactions between the changed code and the rest of the program [14], [15]. Therefore, we aim at using mutation testing using the so-called Commit-relevant mutants [16].

C. Commit-relevant mutants

Commit-relevant mutants are those that make observable any interaction between the altered code and the rest of the program under test. These mutants alter the program semantics that are relevant to the committed changes, i.e., they have behavioural effects on the altered code behaviour. 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. In essence, one can use relevant mutants to capture the 'observable' dependencies between changed and unchanged code [17], [18], which reflect the extent to which test suites are testing the altered program behaviours.

In particular, mutants interact with program changes when the post-commit mutant version (includes both the changes and the mutant) behaves differently from **a)** the related pre-commit mutant version and **b)** the post-commit non-mutated version. These conditions establish that changes and mutants interact [16]. Condition **a)** establishes that the behaviour differences are caused by the presence and absence of the committed changes and condition **b)** that the behaviour differences are caused by the presence and absence of the mutants.

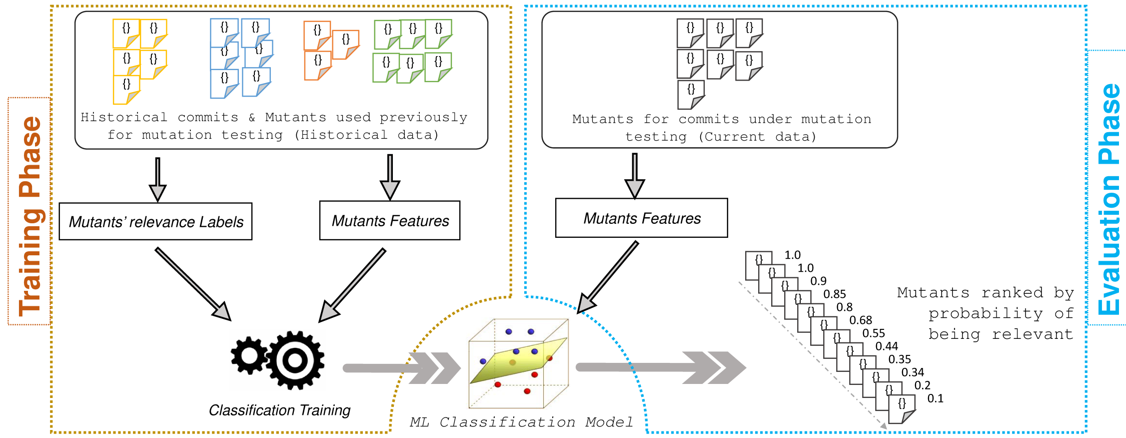

Fig. 1. Overview of *MuDelta*. The learner is trained on a set of mutants from historical code changes that are labeled with respect to given test suites. The machine learner is subsequently used to predict the priority ordering of the set of mutants to identify those most likely to be relevant to a given change.

The virtue of commit-relevant mutation testing, as described in the study of Ma et al. [16] is the best-effort application of mutation testing. This gives the potential for improved fault revelation under the same (relatively low) user effort than using randomly sampled mutants, i.e., traditional mutation testing. However, in order to be useful, these mutants need to be identified in advance, prior to any mutant analysis performed. This is because relevant mutants form the objectives that developers will analyse. To achieve this, we develop a machine learning approach, which we describe in the following section.

Figure 2 presents a commit-relevant mutant on a fault-introducing commit of GNU Coreutils[2]. This is the commit with ID 8 from CoREBench [19]. The commit affects two functions of the program *seq* (*main* and *seq_fast*). The entry-point is the function *main*, which, calls the functions *print_numbers* and *seq_fast* to compute and print the results. The function *seq_fast* is an optimized implementation of the function *print_numbers*, used only when the inputs meet specific conditions. In Figure 2, the line 543 checks the condition to call *seq_fast*. If the condition is satisfied, *seq_fast* is called. Otherwise, *print_numbers* is called. Note that *print_numbers* may be called after *seq_fast* if the later fails (the condition at line 405 is not satisfied, i.e. $a > b$). In that case, the execution of *seq_fast* does not alter the program state or output.

The commit aims at relaxing the condition that guards the call to function *seq_fast*. In the pre-commit version, *seq_fast* is not called when the user specifies a separator. However, in the post-commit version, *seq_fast* is called whenever a) the user specifies a separator, and b) the separator string has a single character.

In the function *seq_fast*, the commit only replaces the hard coded separator ('\n') with separator's global *string* variable. In the function *main*, the commit relaxes the "if" condition at line 543, in a way that *seq_fast* is also called when the user specifies a separator, which can be any single "8-bits" character (it is not limited to '\n').

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

The program *seq* calls *seq_fast* to print all the integers from the first parameter a to the second parameter b, and using a given character (first character of *separator* in post-commit and '\n' in pre-commit) to separate the printed numbers.

Let four mutants such that: mutant M_1 deletes the statement at line 414, which prints the first number using *puts*. Mutant M_2 deletes the modified statement at line 420, which add the separator to the buffer to print. Mutant M_3 swaps the operands of the last "&&" operation at the modified line 543. Mutant M_4 replaces the exit value at line 595 by -1.

We observe that M_4 is not relevant to the commit. In fact, there is no test that can kill M_4 in the post-commit version, and create an output difference between pre- and post-commit versions of M_4. If a test kills post-commit M_4, it must avoid executing line 547, thus, *seq_fast* is either not called or its call does not succeed (does not print anything). Thus, the output of the execution of the pre- and post-commit versions of M_4 with such test will be same (both computed with *print_numbers*, which is not altered by the commit). Mutant M_3 is equivalent, because no clause has side effect that is controlled by another clause in the *if* condition.

However, M_1 is relevant to the commit. An execution of the test "seq -s, 1 2", which sets the separator to the comma (','), outputs "1,2\n" in pre-commit M_1 (*print_numbers* is called), "2," in post-commit M_1 ('puts(p)' is deleted and *seq_fast* is called), and "1\n2," in the post-commit original version (the first number is printed using 'puts(p)', which appends an '\n'). Similarly, M_2 is relevant to the commit. The execution of same test "seq -s, 1 2" outputs "1,2\n" in pre-commit M_2 (*print_numbers* is called), "1\n2" in post-commit M_1 (no comma separator printed and *seq_fast* is called).

Moreover, a fault introduced by the commit makes the program use '\n' instead of the user specified separator, after printing the first number, when the user separator is a single character other than '\n'. This happens because in such scenario, the program calls *seq_fast*, which calls 'puts(p)' (line 414) to print the first number. This automatically add an extra '\n' and do not use the specified separator.

```
390  static bool seq_fast (char const *a, char const *b) {
       …
404    bool ok = cmp (p, p_len, q, q_len) <= 0;
405    if (ok) {
         …
414      puts (p);                          // Mutant M1: delete Statement
         …
418        incr (&p, &p_len);
419        z = mempcpy (z, p, p_len);
420  -     *z++ = '\n';
420  +     *z++ = *separator;
421        if (buf_end - n - 1 < z) {
423          fwrite (buf, z - buf, 1, stdout);     Mutant M2: delete
424          z = buf;                              statement
425        }
         …
433    }
       …
437    return ok;
438  }
.........................................................................
450  int main (int argc, char **argv) {
     …
543  -  if (… && all_digits_p (argv[1]) & …) {
543  +  if (all_digits_p (argv[optind]) && …  && strlen (separator) == 1) {
          …
546       if (seq_fast (s1, s2))            Mutant M3:
547         exit (EXIT_SUCCESS);            Swap operands of "&&".
550     }
        …
592     print_numbers (format_str, layout, first.value, …);
594     exit (EXIT_SUCCESS);  // Mutant M4: EXIT_SUCCESS → -1
595  }
```

- Mutants M1 and M2 are relevant. Moreover, they are 99% fault revealing (99% of the tests killing them find the introduced fault).
- Mutants M3 and M4 are not relevant (M3 is equivalent).

Fig. 2. Mutation testing in a fault introducing commit. The fault is triggered by the call to 'puts(p)', which automatically uses '\n' as the first separator, resulting in not using the user specified separator when this is a single character other than '\n'. This makes every test executing *seq_fast* with a separator other than '\n' to reveal the fault. Killing M_1 or M_2 can result in such tests, while killing M_4 does not (to kill M_4 a test must avoid executing line 547, which means that *seq_fast* is either not called or its call does not print anything, hence not making any observable difference). M_3 is equivalent.

Every test that executes *seq_fast*, with a separator other than '\n' reveal the fault. These are $(1 - \frac{2}{257}) \approx 99.2\%$ of all the tests that successfully execute *seq_fast*. The reason is that the separator is either not set in the test (defaults to '\n'), or set to one of the 256 '8-bits' characters (including '\n'). We observe that all tests that successfully execute *seq_fast* kill M_1 and M_2. Therefore, 99.2% of the tests that kill M_1 and M_2 reveal the fault.

III. Approach

We aim at testing commits using commit-relevant mutants; the subset of mutants on the post-commit program version that has a behaviour relevance to the committed changes [16].

We develop *MuDelta*, a technique that learns to rank mutants according to their commit-relevance potential (likelihood to be commit-relevant). Initially, *MuDelta* applies supervised learning on a mutant corpus from past data, and builds a prediction model. This model is then applied to predict the mutants that should be used to test the future commits of the program under test. This means that at commit time, testers can use and focus only on the most relevant mutants. This process is depicted in Figure 1.

A. MuDelta *Feature Engineering*

The mutant selection process in *MuDelta* is based on training of a predictor that is capable of identifying whether a mutant is commit-relevant with a certain confidence (probability). Consequently, we design a set of features to reflect specific code properties which may discriminate a commit-relevant mutant from another.

The study of Chekam et al. [20] found that fault revealing and killability mutant characteristics can be captured by simple code features. Therefore, we consider the features that they proposed in our machine learning model. Unfortunately, these features do not capture the interaction between mutants and the altered code. Hence, we design additional features capable of capturing the link between the mutant and the altered code (by the commit). These features also aim at capturing the characteristics of the altered code.

In the following subsections we describe the features we use in order to train a classifier. We consider a commit modification C associated with code statements $SC = \{S_{C_1}, S_{C_2}, ..., S_{C_n}\}$, and let $BC = \{B_{C_1}, B_{C_2}, ..., B_{C_k}\}$ the control-flow graph (CFG) basic blocks associated to the statements SC. Let us also consider a mutant M associated to a code statement S_M on which the mutation was applied. Let B_M be the CFG basic block associated to a mutated statement S_M containing the mutated expression E_M.

B. *Contextual Features*

In order to capture contextual information for each program statement, within a program version, we design features that leverage graph analysis technologies. We construct graph representations of the program, where the nodes are the statements of the program, and the edges are various types of relationships between statements. We consider the following four relationships (edge types): data dependency (direct data dependency, indirect data dependency) [21], control dependency, and control flow. Direct data dependency refers to variable value dependency, while indirect data dependency refers to pointer dereference value dependency (the data is accessed through dereferencing a pointer). In total we use the following 6 different graph representations, i.e., **1)** *Utility Graph* (UG) that includes all four edge types we discussed, **2)** *Dependency Graph* (DG) that includes all three dependency edges types, **3)** *Direct Data Dependency Graph* (DDDG) that includes only the direct data dependency edge type, **4)** *Indirect Data Dependency Graph* (IDDG) that includes only the indirect data dependency edge type, **5)** *Control Dependency Graph* (CDG) that includes only the control dependency edge type, and **6)** *Control Flow Graph* (CFG) which includes only the control flow edge type.

For each graph, we leverage graph analysis algorithms to compute a score for each node. We consider the following graph analysis algorithms: Rich-Club coefficient (RCC) [22], [23], Clustering coefficient (CC) [24]–[26], Square Clustering coefficient (SCC) [27], PageRank (PR) [28], and Hits Analysis (HA) [29].

Complexity: Complexity of S_M, approximated by the number of mutants on S_M.
CfgDepth: Depth of B_M according to CFG.
CfgPredNum: Number of predecessor basic blocks, in CFG, of B_M.
CfgSuccNum: Number of successors basic blocks, in CFG, of B_M.
AstNumParents: Number of AST parents of E_M.
NumOutDataDeps: Number of mutants on expressions data-dependent on E_M.
NumInDataDeps: Number of mutants on expressions that E_M is data-dependent.
NumOutCtrlDeps: Number of mutants on statements control-dependents on E_M.
NumInCtrlDeps: Number of mutants on expressions that E_M is control-dependent
NumTieDeps: Number of mutants on E_M.
AstParentsNumOutDataDeps: Number of mutants on expressions data-dependent on E_M's AST parent statement.
AstParentsNumInDataDeps: Number of mutants on expressions that E_M's AST parent expression is data-dependent.
AstParentsNumOutCtrlDeps: Number of mutants on statements control-dependent on E_M's AST parent expression.
AstParentsNumInCtrlDeps: Number of mutants on expressions that E_M's AST parent expression is control-dependent.
AstParentsNumTieDeps: Number of mutants on E_M's AST parent expression.
TypeAstParent: Expression type of AST parent expressions of E_M.
TypeMutant: Mutant type of M, transformation rule. E.g., $a + b \to a - b$.
AstChildHasIdentifier: AST child of expression E_M has an identifier.
AstChildHasLiteral: AST child of expression E_M has a literal.
AstChildHasOperator: AST child of expression E_M has an operator.
OutDataDepNumStmtBB: Number of CFG basic blocks containing an expression data-dependent on S_M.
InDataDepNumStmtBB: Number of CFG basic blocks containing an expression on which S_M is data-dependent.
OutCtrlDepNumStmtBB: Number of CFG basic blocks containing an expression control-dependent on S_M.
InCtrlDepNumStmtBB: Number of CFG basic blocks containing an expression on which S_M is control-dependent.
AstParentMutantTypeNum: Number of each mutant type of E_M's AST parents.
OutDataDepMutantTypeNum: Number of each mutant type on expressions data-dependents on E_M.
InDataDepMutantTypeNum: Number of each mutant type on expressions on which E_M is data-dependent.
OutCtrlDepMutantTypeNum: Number of each mutant type on statements control-dependents on E_M.
InCtrlDepMutantTypeNum: Number of each mutant type on expressions on which E_M is control-dependent.

Fig. 3. Mutant utility features

.

Overall, we get a set of features F_S, for each statement S and for each graph G, by computing the score of the node corresponding to S, using all graph analysis algorithms on G. This gives us 6 * 5 (graphs * Metrics) features per program statement.

C. Mutant utility features

We used the features proposed by Chekam et al. [20]. These features relate to the complexity of the mutated statement S_M, the position of S_M in the control-flow graph, the dependencies with other mutants, and the nature of the code block B_M where S_M is located. The selected features are recorded in Figure 3. Note that for this study, we added the last 9 features (marked in the figure with italic), and the contextual features of S_M (Section III-B). The first 4 features (with italic) are similar to the features *NumOutDataDeps, NumInDataDeps, NumOutCtrlDeps, NumInCtrlDeps* used by Chekam et al. [20], but, instead of the number of mutants, they count the number of basic blocks.

D. Mutant-Modification Interaction Features

To capture the interaction between mutant and altered code, we use features related to the information flow that the altered code C incur to the execution of mutant M. In this regard, we propose features that characterize the altered code and features that capture the information flow between C and M.

NumConditional: Number of conditional statements in the modification.
NumHunks: Number of hunks (blocks) in the commit diff.
HasExit: The modification involves program termination commands.
ChangesCondition: The modification involves the condition of an *if* or a loop.
InvolesOutput: The modification involves a function call to *printf* or *error*.
IsRefactoring: The modification only does code refactoring.
NumUPDATE: Number of UPDATE operations from GumTree tool [30].
NumINSERT: Number of INERT operations from GumTree tool [30].
NumMOVE: Number of MOVE operations from GumTree tool [30].
NumDELETE: Number of DELETE operations from GumTree tool [30].
NumActionClusters: Number of action clusters from GumTree tool [30].
NumActions: Number of actions from GumTree tool [30].

Fig. 4. Mutant-Modification Interaction Features

1) Modification Characteristics Features: We have features extracted from the commit diff and features extracted from the changed or added statements in the post-commit version of the program. Figure 4 describes the features extracted from the commit diff. The features extracted from the changed or added statements are: (a) The mean of the depth, according to CFG, of the basic blocks in BC (*modificationCfgDepth*). (b) The mean of the complexity of the statements in SC (*modificationComplexity*). (c) The contextual features (see Section III-B) of the added or changed statements in the program. When the modification involves multiple statements, the mean of each feature value for all statements is computed.

2) Information-flow Features: The first feature that we use, in this category, is a Boolean variable (*MutantOnModification*) that represents whether the mutant M mutates an altered code ($S_M \in SC$). Additionally, we consider the 6 graphs presented in section III-B, and compute, for each graph, the set of shortest paths between S_M and SC.

For every set of paths, we compute the size (*NumPaths*), the maximum path length (*MaxPathLen*), minimum path length (*MinPathLen*) and mean path length (*MeanPathLen*). Our features are thus, the combination of each one of these metrics on every shortest path set.

E. Implementation

We implemented *MuDelta* in Python. For learning, we used stochastic gradient boosting [31] (decision trees), which has been found to work well in the context of mutation [20]. We used the XGBoost [32] framework and set the number of trees to 3,000 with a maximum trees depth to 10. We adopt early stopping during training to avoid over-fitting.

MuDelta uses both numerical or categorical features. The categorical features are: *TypeAstParent, TypeMutant*. In order to use the feature values with XGBoost, we pre-process them using a normalization of numerical and an encoding of categorical features. We normalize numerical features, between 0 and 1 using *Rescaling* (also known as min-max normalization).

We use *binary encoding* (binary encoding helps to keep a reasonably low feature dimension, when comparing to one-hot-encoding) for the categorical features. We also use NetworkX[3] in the graph representation in order to extract the contextual features that were described in section III-B.

[3]https://networkx.github.io/

IV. Research Questions

We start our analysis by investigating the prediction ability of our machine learning method. Thus, our first research question can be stated as:

RQ1 *(Prediction performance):* How well does *MuDelta* predict commit relevant mutants?

To answer this question we collect a set of commits from the subject programs where we apply mutation testing and identify relevant mutants. Then, we split the commits into training/validation (80% of the commits) and test sets (20% of the commits) based on the timeline of the project(older commits are used for training and newer for commits are used for evaluation), and perform our experiment.

After checking the performance of the predictions, we turn our attention to the primary problem of interest; mutant ranking. We investigate the extent to which our predictions can lead to strong and relevant tests (by using the predictive mutants as test objectives) in contrast to baseline mutants, i.e., randomly sampled mutants among those residing in the changed components (*Random*) or among those residing on the altered lines (*Modification*). Hence we ask:

RQ2 *(Test assessment):* How *MuDelta* compare with the baseline mutant sets with respect to killing commit-relevant mutants?

We answer this question following a simulation of a testing scenario where a tester analyse mutants in order to generate tests [12], [33]. We are interested in the relative differences between the subsumming relevant mutation score, denoted as $rMS*$, when test generation is guided by the predicted or the baseline mutants. We use the subsumming relevant mutation score to avoid bias from trivial/redundant mutants [10]. We also use the random mutant selection baseline since it performs comparably to the state-of-the-art [12], [20], [34]. We compare with random on a best effort basis, i.e., the $rMS*$ 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 [9], [12] and aims at quantifying the benefit of one method over the other. To further show the need for mutant selection out of the changed code, we also compute the extend to which mutants on modification are sufficient in killing commit-relevant mutants.

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

RQ3 *(Fault Revelation):* How *MuDelta* compare with the baseline mutant sets with respect to (commit-introduced) fault revelation?

To answer this question, we investigate the fault revelation potential of the mutant selection techniques based on a set of real fault-introducing commits. We follow the same procedure as in the previous research questions.

TABLE I
Test Subjects

Benchmark	#Programs	#Commits	#Mutants	#Relevant	#Tests
CoREBench	6	13	154,396	21,597	8,828
Benchmark-1	17	37	412,060	65,982	14,785

V. Experimental Setup

A. Benchmarks Used

We selected C programs from the GNU Coreutils[4], a collection of text, file and shell utility programs widely used in software testing research [19], [35], [36]. The whole codebase of Coreutils comprises approximately 60,000 lines of C code[5]. To perform our study on commits we used the benchmark[6] introduced by Ma et al. [16] that is composed of two parts and includes *Benchmark-1*, a set of commits mined from the Coreutils' Github repository from year 2012 to 2019 and *CoREBench* [19] that has fault introducing commits.

The benchmark contains a) mutants generated by Mart [37], a state-of-the-art tool that supports a comprehensive set of mutation operators and TCE[7] [9], [38] on both pre- and post-commit program versions of each commit, b) the mutant labels (whether they are commit-relevant), and c) large test pools created using a combination of test generation tools [35], [36], [39]. It is noted that the mutant test executions involved require excessive computational resources, i.e., require roughly 100 weeks of computation. Details about the data we used are recorded in Table I. The column *#Relevant* records the number of commit-relevant mutants.

B. Experimental Procedure

To account for our working scenario, we always train according to time, i.e, we use the older commits for training and the newer for evaluation. This ensured that we follow the historical order of the commits.

Following the stated RQs, our experiment is composed of three parts. The first part evaluates the prediction ability (performance) of *MuDelta*, answering RQ1. The second at evaluating the ability of *MuDelta* to rank commit-relevant mutants, answering RQ2, and the third part at evaluating the fault revealing potential, answering RQ3.

First experimental part: We evaluate the trained classifiers using five typically adopted metrics, namely, the Area Under the Receiver Operating Characteristic Curve (ROC-AUC), the Area Under the Precision-Recall Curve (PR-AUC), the precision, the recall and the F1-score.

The Receiver Operating Characteristic (ROC) curve records the relationship between true and false positive rates [40]. The Precision-Recall (PR) Curve records the decrease in true positive classifications when the predicted positive values increase. In essence, the PR curve shows the trade-off between precision and recall [40].

[4] https://www.gnu.org/software/coreutils/
[5] Measured with cloc (http://cloc.sourceforge.net/)
[6] https://github.com/relevantMutationTesting
[7] Compiler-based equivalent and duplicate mutant detection technique

Precision is defined as the number of items that are truly relevant among the items that predicted to be relevant. *Recall* is defined as the number of items that are predicted to be relevant among all the truly relevant ones. The F1-score or F-measure of a classifier is defined as the weighted harmonic mean of the precision and recall. These assessment metrics measure the general classification accuracy of the classifier. Higher values denote a better classification.

To reduce the risk of over-fitting, we split our commit data into three mutually exclusive sets (training, validation and test data). We also use early stopping during training to overwhelm over-fitting. We use the following procedure:

1) Chronologically order the commit (from older to newer).
2) Select the newest 20% of commits as test data.
3) Randomly shuffle all the mutants from the remaining 80% of commits (oldest commit), then, select 20% of them as validation data and the rest as training data.

Thus, the training, validation and test data represent 64%, 16% and 20% of the data-set, respectively. The model evaluation is performed on the test data. This experiment part was performed on both CoREBench and *Benchmark-1*.

Second experimental part: We simulate a scenario where a tester selects mutants and designs tests to kill them. This typical procedure [1], [12], [16], [20], [41] consists of randomly selecting test cases, from the test pools of the benchmark, that kill the selected mutants. Specifically, we rank the mutants and then we follow the mutant order by picking test cases, from the test pool, that kill them. We then remove all the killed mutants and pick the next mutant from the list. If the mutant is not killed by any of the tests, we discard it without selecting any test. We repeat this process 100 times for all the approaches. *MuDelta* ranks all the mutants by the predicted commit-relevance probability, *Random* randomly ranks all the mutants in the changed components, and *Modification* randomly ranks the mutants located on the altered code.

Our effectiveness metrics are the relevant subsuming mutation score ($rMS*$) achieved by the test suites when analysing up to a certain number of mutants. Subsuming score metrics allows reducing the influence of redundant mutants [10], [13], [42]. We also compute the Average Percentage of Faults Detected (APFD) [43] that represents the average relevant subsuming mutation score when analysing any number of mutants within a given range.

Our effort metric is the number of mutants picked (analysed by the tester). This includes the mutants, killable or not, that should be presented to testers for analysis (either design a test to kill them or judge them as equivalent) when applying mutation testing [9], [12]. In the spirit of the best-effort evaluation, we focus on few mutants (up to 100) that testers need to analyse. This evaluation aims at showing the benefits of *MuDelta* over *Random* under the same relative testing effort. The contrast with the *Modification* shows whether there is a need for mutant selection outside of the modified code, i.e., whether mutants on modification are sufficient leading to tests that kill commit-relevant mutants. This part of the experiment was performed on both CoREBench and *Benchmark-1*.

Third experimental part: To evaluate the fault revealing ability of *MuDelta*, we used the CoREBench commits. We adopted a chronological ordering for training, validation and testing when splitting the commits similar to what we did in previous experimental parts. We use the same process and effort metric as in the the second part of the experiment and report results related to fault revelation and the average percentage of commit-introduced faults revealed (APFD) within the range, 1-100, of analysed mutants.

To account for the stochastic selection of test cases and mutant ranking, we used the Wilcoxon test to determine whether there is a statistically significant difference between the studied methods. To check the size of the differences we used the Vargha Delaney effect size $\hat{A}_{12}$ [44], which quantifies the differences between the approaches. A value $\hat{A}_{12} = 0.5$ suggests that the data of the two samples tend to be the same. Values $\hat{A}_{12} > 0.5$ indicate that the first data-set has higher values, while values $\hat{A}_{12} < 0.5$ indicate the opposite.

VI. Results

A. Assessment of the Prediction Performance (RQ1)

To evaluate the performance of *MuDelta*, we check the model's convergence. During training and after each iteration of the training process, we check the model performance on both the training and validation data we used for training. Figure 5 shows the ROC-AUC and PR-AUC values wrt the number of training iterations. We observe that the model performance on both the training and validation data increase with the number of iteration and stabilizes at specific values, suggesting that our model is able to learn the characteristics of commit-relevant mutants.

We then evaluate the performance of our model to predict commit-relevant mutants on the future commits that appear in the test set. To compute the precision, recall and F1-score, we set the prediction threshold probability to 0.1, which we obtained by applying the geometric mean [45], [46] on the validation dataset. The precision, recall and F1-score of our classifier are 0.63, 0.32 and 0.42, respectively. These values are higher than those that one can get with a random classifier (0.21, 0.21 and 0.20, respectively). Figure 6 shows the ROC and PR curves of our classifier (strong lines) and a random classifier (dashed lines). We observe that the ROC-AUC of our classifier is 0.80 indicating a strong prediction ability. Similarly, we see that the PR-AUC of our classifier is 0.50 while the random classifier PR-AUC is 0.20.

In this context [7] it is important to give few mutants to developers for analysis. To evaluate the performance of *MuDelta* with lower thresholds, we also study the performance of *MuDelta* with thresholds ranging from the 10 to 100 mutants. We observe that the median precision of *MuDelta* ranges from 0.76 to 0.90 when the threshold goes from 10 to 30 mutants. These values are significantly higher than the random classifier, which has a precision of 0.15.

These results provide evidence that *MuDelta* provides a good discriminative ability for assessing the utility of mutants to test particular code changes.

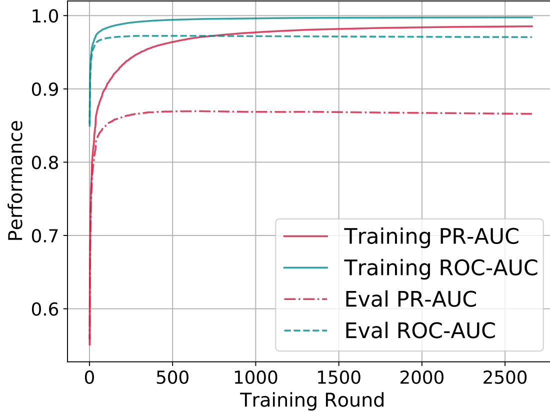

Fig. 5. Training and Validation Curves from the Training phase.

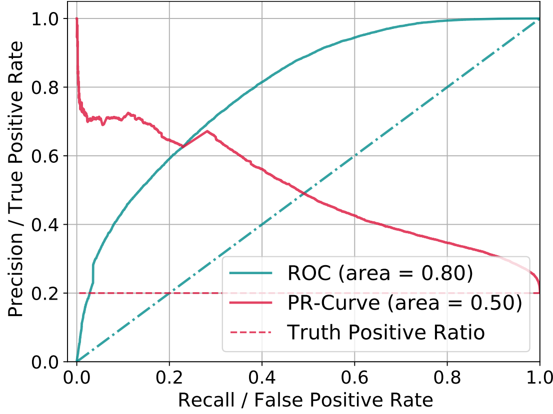

Fig. 6. Precision-Recall and ROC Curves on test data.

B. Mutant Ranking for Tests Assessment (RQ2)

Figure 7 shows the median $rMS*$ achieved by the mutant ranking strategies, when the number of analysed mutant budget range from 1 to 100 mutants. In other words, the figure shows test effectiveness (measured with $rMS*$, y-axis) that is achieved by a developer when analysing a number of mutants, representing the cost factor (recorded in x-axis). Each sub-figure is a commit taken from the test data. We observe that the curve for *MuDelta* is always higher than the curves of *random* and *Modification*, and *Random* is above *Modification*.

To further visualize the differences, Figure 8 shows the distribution of the $rMS*$ of the mutant ranking strategies for budget thresholds 10, 30, 50 and 100 mutants. As can be seen from the plots, *MuDelta* outperforms both *Random* and *Modification*. Interestingly, *Random* outperforms *Modification*. With threshold 10 mutants, the difference of the median values is 22% and 26% for *Random* and *Modification*, respectively. This difference is markedly increased when analysing more mutants, i.e., it becomes 45% and 50% for the thresholds of 30 and 50 mutants, for *Random*.

To check whether the differences are statistically significant we performed a Wilcoxon rank-sum test and computed the Vargha Delaney $\hat{A}_{12}$ effect size and found that *MuDelta* outperforms both *Random* and *Modification* with statistically significant difference (at 0.01 significant level). *Random* has also statistically significant differences with *Modification*.

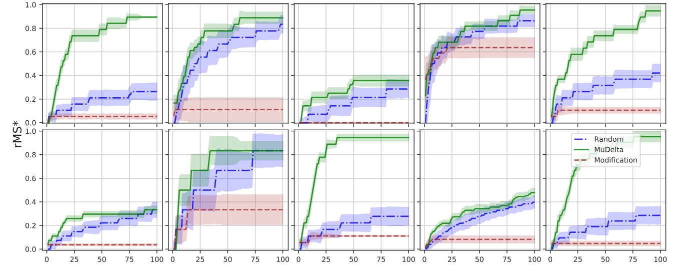

Fig. 7. $rMS*$ achieved when analysing up to 100 mutants.

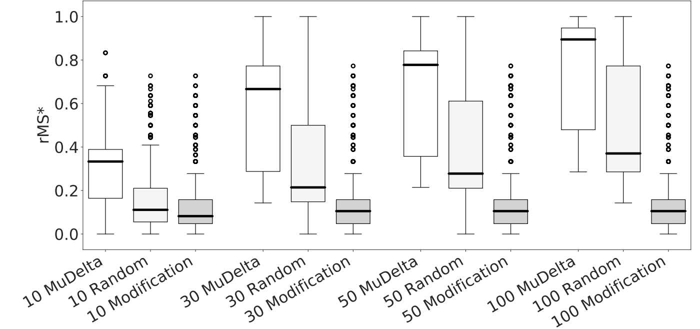

Fig. 8. $rMS*$ values when analysing up to 10, 30, 50 and 100 mutants.

Figure 9 shows the Vargha Delaney $\hat{A}_{12}$ values between *MuDelta* and both *Random* and *Modification*. We observe that the median value is between 77% and 83% for threshold between 10 and 100 mutants, for *Random*. Suggesting that *MuDelta* is better than *Random* in 77% to 83% of the cases for these thresholds. The differences are larger for *Modification*.

We further validate our approach by considering the distributions of APFD (Average Percentage of Faults Detected) values for all possible thresholds (for 1-100 mutants). Figure 10 depicts these results and shows that *MuDelta* yields an APFD median of 71%, *Random* and *Modification* reach median APFD values of 26% and 11% respectively, confirm the superiority of our approach.

To account for the stochastic nature of the compared approaches and increase the confidence on our results, we further perform a statistical test on the APFD values. The Wilcoxon test results yielded p-values much lower than our significance level for the compared data, i.e., samples of *MuDelta* and *Random*, *MuDelta* and *Modification*, *Random* and Modification, respectively. Therefore, we conclude that *MuDelta* outperforms *Random* with statistically significance, while *Modification* is not sufficient for testing the deltas.

C. Mutant Ranking and Fault Revelation (RQ3)

Figure 11 shows the distributions of APFD (Average Percentage of Faults Detected) values for the CoREBench fault introducing test commits, using the three approaches under evaluation. While *MuDelta* yields an APFD median of 52%, *Random* and *Modification* reach median APFD values of 25% and 0% respectively. The improvement over *Random* and *Modification* are 27% and 52%, respectively. These results confirm the superiority of our approach wrt to fault revelation.

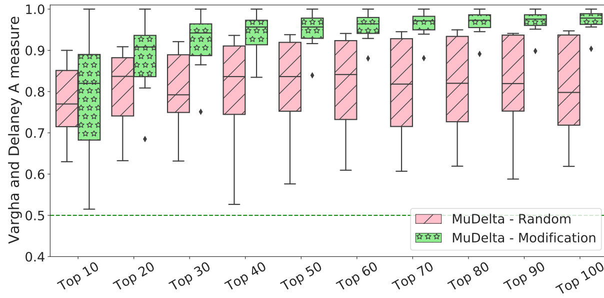

Fig. 9. Vargha and Deianey $\hat{A}_{12}$ (*MuDelta* VS Random, *MuDelta* VS Modification) about $rMS*$

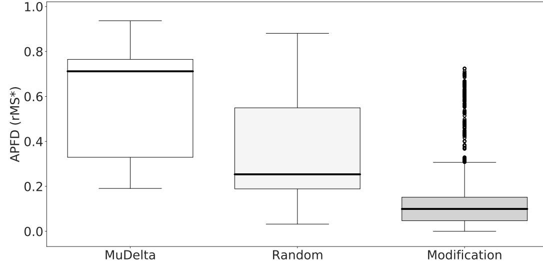

Fig. 10. APFD $rMS*$ (up to 100 mutants).

The Wilcoxon test yielded p-values much lower than our significance level for the compared data, i.e., samples of *MuDelta* and *Random*, *MuDelta* and *Modification*, *Random* and *Modification*. Therefore, we conclude that *MuDelta* outperforms *Random* and *Modification* with statistically significance while *Random* outperforms *Modification*.

Figure 12 shows the distribution of fault revelation for the ranking strategies and for mutant set size thresholds up to 100 mutants. We observe that the curve for *MuDelta* is above the curves of *random* and *Modification*, and *Random* is above *Modification*. Specifically, we observe that *MuDelta* reaches a fault revelation of 60% and 100% when analysing the top 30 and 61 mutants, while *Random* 7% and 12%, respectively.

VII. Discussion

A. Comparison with other models

To further assess the effectiveness of our model, we contrast it with the prediction ability of five other models (on the same training, validation and test data-sets) that are typically used in prediction modelling studies. In particular, we used three families of models (Ensemble model classifiers, Logistic classifiers and Neural Networks) and built five models; namely Adaboost, Random Forest, Logistic Regression, Multilayer Perceptron (MLP) and Mixed MLP. MLP and Mixed MLP were inspired by the work of Li et al. [47], their architecture is shown in Figure 13 and 14. To train and evaluate the models we used the Sklearn library[8]. Since our data are imbalanced we also used class weighting strategies that are commonly used to tackle this issue. To avoid bias from improper setting of the learners, in all the cases we used Grid Search Cross Validation on the validation set to tune our hyperparameters.

[8]https://scikit-learn.org/stable/

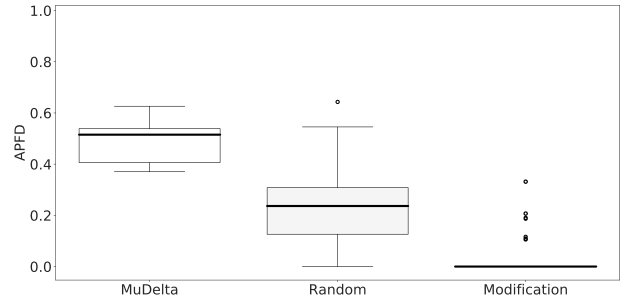

Fig. 11. APFD Fault-revelation (up to 100 mutants).

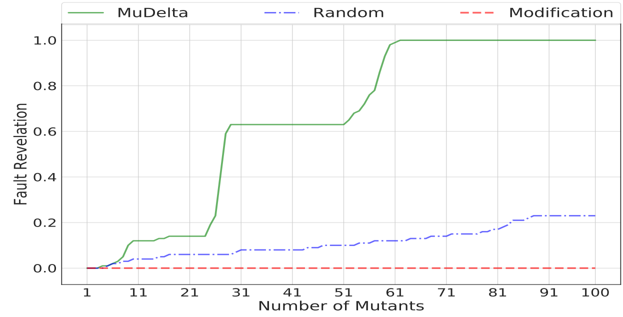

Fig. 12. Median fault-revelation in fault introducing commits.

TABLE II
Model Comparison

	ROC-AUC	PR-AUC	MCC	Precision on Top-100
AdaBoost	0.6	0.35	0.26	0.55
Random Forest	0.66	0.31	0.24	0.57
Logistic	0.58	0.26	0.13	0.21
MLP	0.51	0.19	0.1	0.2
Mixed MLP	0.68	0.45	0.31	0.2
XGBoost	*0.80*	*0.50*	*0.36*	*0.61*

Table II reports the ROC-AUC, PR-AUC, MCC, and precision on top 100 ranked mutants of the prediction results of all different learners we built. The results show that the XGBoost model, that we use, perform best in all cases. The general prediction metrics (ROC-AUC,PR-AUC, MCC) show that Mixed MLP model is the second best case though it falls behind the Ensemble models wrt to the top-100 mutants. Nevertheless, the results provide clear indications that the XGBoost model we use is indeed the best choice.

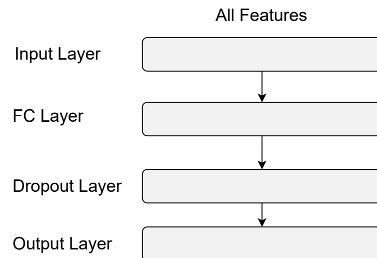

Fig. 13. MLP - Neural Network Architecture

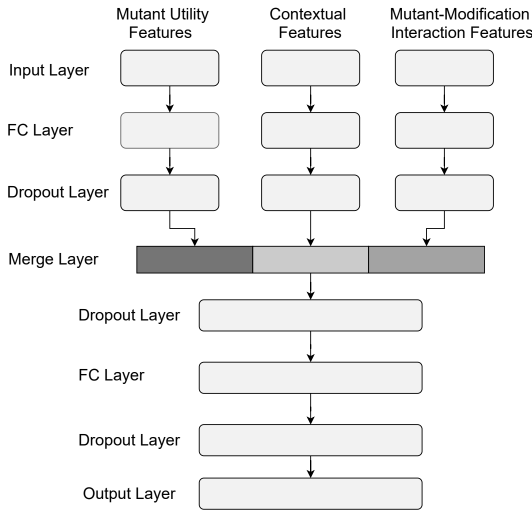

Fig. 14. Mixed MLP - Neural Network Architecture

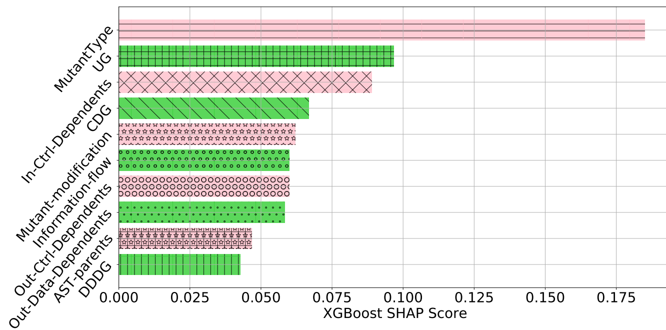

Fig. 15. Feature Importance, SHAP Score of top-10 feature sets. The 10 most important features are "Mutant Type", "Utility Graph", "Incoming Control Dependencies", "Mutant-modification features", "Control Dependency Graph", "Information-flow", "Outgoing Control Dependencies", "Outgoing Data Dependencies","AST parents" and "Directed Data Dependency Graph".

B. Feature Importance

To evaluate the importance of our features we used the SHapley Additive exPlanations (SHAP)[9] method [48], i.e., a game theory method that explains individual predictions based on the game theoretically optimal Shapley Values. In particular, we aim at explaining our predictions by assuming that each feature value we use is a "player" in a game where the prediction is the payout. Shapley values – a method from coalitional game theory – tells us how to fairly distribute the "payout" among the features. We thus measure and report the feature importance (Shapley values) of the feature categories we use. Results are depicted on Figures 15 and show that "Mutant Type", "Utility Graph", "Incoming Control Dependencies", "Mutant-modification features", "Control Dependency Graph" and the "Information-flow" are the top 6 feature sets and that all three types of features we use are important. Additional results related to the feature importance of the individual features we used can be found on the accompanied website.

[9]https://github.com/slundberg/shap

VIII. Threats to Validity

A possible threat to external validity could be due to our test subjects. Our target was commits that do not alter test contracts and make small modifications, similar to those observed in industrial CI pipelines. Such commits are usually hard to test and typically result in subtle faults. Large commits that add new features, should be anyway tested by using a mutation testing approach that involves (almost) all the relevant mutants residing on the added code. To reduce this threat, we sampled a commit set where we could reasonably perform our experiments. At the same time, to diminish potential selection bias, we also used the Coreutils commits of CoREBench [19], which are frequently used in testing studies.

We are confident on our results since the relevance properties of the mutants reside on the context of the committed code, which includes the area around the dependencies to the committed code (where we draw our feature values), that is small and its characteristics should be as representative as our subjects. Moreover, our predictions converge well, do not have significant variance wrt to the baselines and consistently outperform the baselines in all test subjects we used. Additionally, the statistical significance we observe indicates the sufficiency of our data analysis [49]. Future work should validate our findings and analysis to larger programs.

Another threat may relate to the mutants we use. To mitigate this threat, we selected data from a mutation testing tool [37] that has been used in several studies [16], [20], [39] that supports the most commonly used operators [50] and covers the most frequent features of the C language.

Threats to internal validity may be due our features. We use a large number of features, selected either based on previous studies [20] or by using our intuition, which are automatically filtered by gradient boosting. To further reduce this concern, we split our data in three parts, training, validation and test data. During training (using training data) we measure the model convergence on training and validation data. As demonstrated in Figure 5, our model converges both on the training and validation data, showing that there are low chances for over- or under-fitting because in these cases, the model would not converge on the validation data.

The test-based approximation of relevant and killable mutants may introduce additional threats. To reduce it, we used test suites generated by KLEE [35] and SeMu [39], together with developer test suites.

A possible threat to construct validity could be due to the effort metric, i.e., the number of analysed mutants, we use. This is a typical metric for this kind of studies [12] aiming at capturing the manual effort involved when analysing mutants or asserting automatically generated tests. Since, our data have been filtered by TCE [9], [38], a state-of-the-art equivalent mutant detection technique, this threat should be limited.

Overall, we tried to reduce threats by using various evaluation metrics, i.e., prediction performance, relevant mutation score and fault revelation, and established procedures. Furthermore, to enable replication and future research we will make our tools and data publicly available.

IX. Related Work

The problem of determining the set of mutants that are most relevant to particular code changes might resemble a dependence analysis problem. One natural solution involves forming a program slice on the set of changed statements. Any mutant that lies in the slice should be considered relevant. Unfortunately, this approach does not scale well for several reasons. Firstly, as have been previously observed [51], [52], even a single static slice of a program tends to occupy between one and two thirds of the program from which it is constructed. Therefore, the union of a set of such slices, will be large, and thereby fail to exclude many mutants. Secondly, the dependence analysis would need to be incremental, which raises further challenges. Although there have been incremental dependence analyses in the literature [53], many well-developed slicing systems are not incremental. In general, the problem of incremental program analysis at scale remains challenging [5]. Thirdly, it is hard to use dependence analysis to provide the priority ordering we need, where priority is based on degree of relevance. Potentially, unions of dynamic slices or some form of observation-based slicing [17] could achieve this, but such approaches have a prohibitive computational cost in comparison to our method.

Change impact analysis [54] aims at determining the effects of changes on the other parts of the software. Similar to program slicing, such approaches are conservative, therefore they result in large number of false positives, does not account for equivalent mutants located on potentially infected code and is hard to provide the mutant ranking (prioritizes mutant types and location) we need. Other attempts aim at testing the potential propagation flows of the changes [14], [15], [55], [56]. Similarly to change impact analysis their purpose is to identify the program paths (flows) that may be impacted by the changes. They rely on symbolic execution to check for the feasibility of the flows, form test requirements (conditions to be fulfilled) and decide on relevance. Unfortunately, such techniques inherit most of the issues of symbolic execution, are complex to implement and test the propagation of the changes. In contrast our technique scales since it relies on static code features, does not require any complex analysis techniques and applies mutation testing that is known for capturing the fault-revealing properties of test suites [1], [20].

Automatic test case generation aims at producing test inputs that a) make observable the code differences of two program versions [57], b) increase and optimize coverage [58] and kill mutants [39], [59], [60]. Among these techniques, the most relevant to our study are the are the ones related to patch testing, i.e., differential symbolic execution [61], KATCH [62] and Shadow symbolic execution [35]. These techniques generate tests exercising the semantic differences between program versions guided by coverage. All these techniques do not propose any test requirements as done by *MuDelta* and thus, they are complementary to our goal. This means that they can be used to generate tests to kill the commit-relevant mutants proposed by *MuDelta*.

Related to continuous integration, Google [7] is using a mutation testing tool that is integrated with the code review process (reviewers select mutants). This tool proposes mutants to developers in order to design test cases. The key basis of this approach is to choose some mutants from the lines of the altered code. We share a similar intent, though we aim at making an informative selection of mutants among all project mutants. According to our results mutants residing on non-altered code tend to be powerful at capturing the interactions between the altered and non-altered code.

Regression mutation testing [63] and the predictive mutation testing [64], [65] also focus on regression testing. Similarly, Pitest [66], a popular mutation testing tool, implements an incremental analysis that computes which mutants are killed or not by a regression test suite. This means that the goal of the above techniques is to estimate the mutation score achieved by regression test suites thereby not making any distinction between commit-relevant and non-relevant mutants, not making any mutant ranking and not proposing any live mutant to be used for test generation.

Fault revealing mutant selection [20] aims at selecting mutants that are likely to expose faults. While powerful, that technique targets the entire program functionality and not the changed/delta one. Since it is unaware of the deltas it selects many irrelevant mutants, while missing many delta-relevant mutants related to the delta-context interactions.

Perhaps the closest work to ours is the commit-aware mutation testing study [16] that defines the notion of mutant relevance and demonstrates its potential. In essence that work describes the fundamental aspects of relevant mutants but does not define any way to identify them at the testing time. We therefore built on top of this notion by providing a static technique that identifies relevant mutants.

Overall, there is a fundamental difference on the aims of our approach and previous research since we statically produce relevant, to code changes, mutants and rank them to provide a best effort testing application.

X. Conclusion

We presented *MuDelta* a delta-oriented mutation testing approach that selects delta-relevant mutants; mutants capturing the program behaviours affected by specific program changes. Experiments with *MuDelta* demonstrated that it identifies delta-relevant mutants with 0.63 and 0.32 precision and recall. Interestingly, killing these mutants leads to strong tests that kill 45% more relevant mutants than killing randomly selected mutants. Our results also show that *MuDelta* selects mutants with a 27% higher fault revealing ability than randomly selected mutants.

Acknowledgement

This work is supported by the Luxembourg National Research Funds (FNR) through the CORE project grant C17/IS/11686509/CODEMATES. Mark Harman is part supported by European Research Council Advanced Fellowship grant number 741278; Evolutionary Program Improvement (EPIC).

References

[1] 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

[2] 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

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

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

[5] M. Harman and P. W. O'Hearn, "From start-ups to scale-ups: Opportunities and open problems for static and dynamic program analysis," in *18th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2018, Madrid, Spain, September 23-24, 2018*. IEEE Computer Society, 2018, pp. 1–23. [Online]. Available: https://doi.org/10.1109/SCAM.2018.00009

[6] 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

[7] 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

[8] 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

[9] 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

[10] 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

[11] 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.

[12] 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

[13] M. Papadakis, T. T. Chekam, and Y. L. Traon, "Mutant quality indicators," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops, Västerås, Sweden, April 9-13, 2018*. IEEE Computer Society, 2018, pp. 32–39. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/ICSTW.2018.00025

[14] 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

[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] W. Ma, T. Laurent, M. Ojdanic, T. T. Chekam, A. Ventresque, and M. Papadakis, "Commit-aware mutation testing," in *Proceedings of the 36th IEEE International Conference on Software Maintenance and Evolution, ICSME*, 2020.

[17] D. W. Binkley, N. Gold, M. Harman, S. S. Islam, J. Krinke, and S. Yoo, "ORBS: language-independent program slicing," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, S. Cheung, A. Orso, and M. D. Storey, Eds. ACM, 2014, pp. 109–120. [Online]. Available: https://doi.org/10.1145/2635868.2635893

[18] M. Kintis, M. Papadakis, and N. Malevris, "Employing second-order mutation for isolating first-order equivalent mutants," *Softw. Test., Verif. Reliab.*, vol. 25, no. 5-7, pp. 508–535, 2015. [Online]. Available: https://doi.org/10.1002/stvr.1529

[19] 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

[20] 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

[21] M. Chalupa, "Slicing of llvm bitcode," *Masaryk Univ*, 2016.

[22] J. J. McAuley, L. da Fontoura Costa, and T. S. Caetano, "Rich-club phenomenon across complex network hierarchies," *Applied Physics Letters*, vol. 91, no. 8, p. 084103, 2007.

[23] R. Milo, N. Kashtan, S. Itzkovitz, M. E. Newman, and U. Alon, "On the uniform generation of random graphs with prescribed degree sequences," *arXiv preprint cond-mat/0312028*, 2003.

[24] J. Saramäki, M. Kivelä, J.-P. Onnela, K. Kaski, and J. Kertesz, "Generalizations of the clustering coefficient to weighted complex networks," *Physical Review E*, vol. 75, no. 2, p. 027105, 2007.

[25] G. Fagiolo, "Clustering in complex directed networks," *Phys. Rev. E*, vol. 76, p. 026107, Aug 2007. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevE.76.026107

[26] J.-P. Onnela, J. Saramäki, J. Kertész, and K. Kaski, "Intensity and coherence of motifs in weighted complex networks," *Physical Review E*, vol. 71, no. 6, p. 065103, 2005.

[27] P. G. Lind, M. C. González, and H. J. Herrmann, "Cycles and clustering in bipartite networks," *Phys. Rev. E*, vol. 72, p. 056127, Nov 2005. [Online]. Available: https://link.aps.org/doi/10.1103/PhysRevE.72.056127

[28] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Technical Report 1999-66, November 1999, previous number = SIDL-WP-1999-0120. [Online]. Available: http://ilpubs.stanford.edu:8090/422/

[29] J. M. Kleinberg, "Authoritative sources in a hyperlinked environment," *J. ACM*, vol. 46, no. 5, p. 604–632, Sep. 1999. [Online]. Available: https://doi.org/10.1145/324133.324140

[30] 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

[31] J. H. Friedman, "Stochastic gradient boosting," *Computational Statistics & Data Analysis*, vol. 38, no. 4, pp. 367–378, 2002.

[32] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 785–794. [Online]. Available: https://doi.org/10.1145/2939672.2939785

[33] 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

[34] R. Gopinath, I. Ahmed, M. A. Alipour, C. Jensen, and A. Groce, "Mutation reduction strategies considered harmful," *IEEE Trans. Reliab.*, vol. 66, no. 3, pp. 854–874, 2017. [Online]. Available: https://doi.org/10.1109/TR.2017.2705662

[35] 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

[36] 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. OSDI'08. USA: USENIX Association, 2008, p. 209–224.

[37] 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

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

[39] 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

[40] A. Zheng, *Evaluating Machine Learning Models A Beginner's Guide to Key Concepts and Pitfalls*. O'Reilly Media, Inc, 2015. [Online]. Available: https://www.oreilly.com/data/free/evaluating-machine-learning-models.csp

[41] A. S. Namin, J. H. Andrews, and D. J. Murdoch, "Sufficient mutation operators for measuring test effectiveness," in *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, Leipzig, Germany, 10-18 May 2008, pp. 351–360.

[42] B. Kurtz, P. Ammann, M. E. Delamaro, J. Offutt, and L. Deng, "Mutant subsumption graphs," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2014.

[43] C. Henard, M. Papadakis, M. Harman, Y. Jia, and Y. L. Traon, "Comparing white-box and black-box test prioritization," in *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, 2016, pp. 523–534. [Online]. Available: http://doi.acm.org/10.1145/2884781.2884791

[44] A. Vargha and H. D. Delaney, "A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong," *Jrnl. Educ. Behav. Stat.*, vol. 25, no. 2, pp. 101–132, 2000.

[45] M. Kubat, S. Matwin *et al.*, "Addressing the curse of imbalanced training sets: one-sided selection," in *Icml*, vol. 97. Citeseer, 1997, pp. 179–186.

[46] R. Barandelaa and E. Rangela, "Strategies for learning in class imbalance problems," 2002.

[47] X. Li, W. Li, Y. Zhang, and L. Zhang, "Deepfl: integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*, D. Zhang and A. Møller, Eds. ACM, 2019, pp. 169–180. [Online]. Available: https://doi.org/10.1145/3293882.3330574

[48] S. M. Lundberg and S. Lee, "A unified approach to interpreting model predictions," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, Eds., 2017, pp. 4765–4774. [Online]. Available: http://papers.nips.cc/paper/7062-a-unified-approach-to-interpreting-model-predictions

[49] A. Arcuri and L. C. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011, Waikiki, Honolulu , HI, USA, May 21-28, 2011*, R. N. Taylor, H. C. Gall, and N. Medvidovic, Eds. ACM, 2011, pp. 1–10. [Online]. Available: https://doi.org/10.1145/1985793.1985795

[50] 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.

[51] D. W. Binkley and M. Harman, "Locating dependence clusters and dependence pollution," in *21st IEEE International Conference on Software Maintenance (ICSM 2005), 25-30 September 2005, Budapest, Hungary*. IEEE Computer Society, 2005, pp. 177–186. [Online]. Available: https://doi.org/10.1109/ICSM.2005.58

[52] D. W. Binkley, M. Harman, and J. Krinke, "Empirical study of optimization techniques for massive slicing," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 1, p. 3, 2007. [Online]. Available: https://doi.org/10.1145/1290520.1290523

[53] A. Orso, S. Sinha, and M. J. Harrold, "Incremental slicing based on data-dependences types," in *International Conference on Software Maintenance (ICSM 2001)*, Los Alamitos, California, USA, Nov. 2001, pp. 158–167.

[54] B. Li, X. Sun, H. Leung, and S. Zhang, "A survey of code-based change impact analysis techniques," *Softw. Test. Verification Reliab.*, vol. 23, no. 8, pp. 613–646, 2013. [Online]. Available: https://doi.org/10.1002/stvr.1475

[55] 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

[56] R. Santelices and M. J. Harrold, "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

[57] 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

[58] 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

[59] 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

[60] B. H. Smith and L. Williams, "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

[61] 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

[62] 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

[63] 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.

[64] 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

[65] 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

[66] 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