Are Mutants a Valid Substitute for Manually-Seeded Faults for Evaluating Student Test Suite Quality?

James Perretta[1], Andrew DeOrio[2], Arjun Guha[1] and Jonathan Bell[1]
perretta.j@northeastern.edu,awdeorio@umich.edu,a.guha@northeastern.edu,j.bell@northeastern.edu
[1]Northeastern University, Boston MA USA
[2]University of Michigan, Ann Arbor MI USA

ABSTRACT

Mutation testing has gained traction as a test suite quality metric that addresses some of the limitations of code coverage. There is evidence in the literature that mutants are a valid substitute for real faults in large open-source software projects. However, it is unclear whether mutants are representative of the kinds of faults that novice programmers make when learning to write software.

A common practice in computer science courses is to evaluate student-written test suites against a set of manually-seeded faults handwritten by an instructor. Writing these faults by hand, however, is a time consuming and potentially error-prone process. If mutants are a valid substitute for faults found in student written code, and if mutant detection is correlated with manually-seeded fault detection, then instructors can instead evaluate student test suites using mutants generated by open-source mutation testing tools.

Following a well-established methodology, we empirically evaluate whether mutants are a valid substitute for manually-seeded faults. We also evaluate whether mutation score is correlated with real student fault detection. Our dataset includes a total of 2,711 assignment submissions across six programming assignments from a total of four courses at three different R1 institutions. We collected one submission per student per assignment. Our results show a strong correlation between mutation score and manually-seeded fault detection rate and a moderately strong correlation between mutation score and real student fault detection. We find that mutants generated from multiple implementations of the same specification are likely to represent more real faults than those generated from a single implementation. Our findings have implications for software testing researchers, educators, and tool builders alike.

ACM Reference Format: James Perretta, Andrew DeOrio, Arjun Guha and Jonathan Bell. 2022. Are Mutants a Valid Substitute for Manually-Seeded Faults for Evaluating Student Test Suite Quality?. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2022).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Testing is one of the most important ways to ensure that software behaves correctly, and one of the most common testing strategies is the use of human-written test suites. Writing test suites by hand is necessitated by the complexity of the oracle problem: While some oracles are simple properties like “the program should not crash,” human input is needed to specify the full range of desired and undesired program behaviors. Since it is still considered best-practice for software to be tested with a suite of human-written test cases, computer science students should be taught how to effectively test their own software.

There is a growing body of work that discusses how to teach software testing and how to evaluate student-written test cases. Early work focused on using code coverage as both an evaluation metric and feedback mechanism [6]. One major limitation of code coverage, however, is that it does not guarantee that the assertions in a test suite properly validate program behaviors [2, 20]. Some instructors use an “all-pairs” approach where every student-written test suite is run against every other student-written implementation [7]. While this strategy has the benefit of evaluating student test suites against real faults in student code, it takes significant manual and computational effort to apply fairly and accurately [31]. The effort required to address these challenges increases super-linearly as the number of students increases.

Other instructors choose to write a set of manually-seeded faults (applied to an instructor-written implementation) and evaluate how many of those manually-seeded faults each student-written test suite detects [31]. This strategy gives the instructor full control over the number and type of faults used to evaluate student test suites but still requires significant manual effort. Additionally, manually-seeded faults applied to an instructor-written implementation may not be representative of the full range of student faults, as students tend to approach problems in fundamentally different ways than experts [3, 29].

An alternative to writing manually-seeded faults is to seed them automatically (mutation testing), a practice that is gaining adoption in industry [16] and open-source software projects [25]. Prior work has explored using mutation testing to evaluate student-written test suites [2, 19, 24], but there is no evidence that mutation testing is an effective stand-in for instructor-written faults when grading student test suites. There is evidence in software engineering research that automatically-seeded faults (mutants) are a valid substitute for real faults in large open-source software projects [12], but it is unclear of whether these results also apply to student-written code. Prior work suggests that student-written code might have different kinds of faults than expert-written code, and hence it is unclear whether mutants are a valid substitute for student faults. Making matters more confusing, prior work has reached conflicting conclusions on the question of whether mutation testing an effective means of evaluating student test suite quality [7, 20, 24].

Contributions: In this paper, we examine the question: "is mutation testing an effective means of evaluating student test suites." We conduct a large-scale empirical evaluation of student test suites, following the well-established methodology of Just et al [12]. Unlike prior studies of mutation efficacy that examine faults in multiple revisions of the same implementation, our study provides new insights by examining multiple *independently developed* implementations of the same specification. Our datasets include a total of 2,711 assignment submissions across six programming assignments from a total of four courses at three different R1 institutions. We collected one submission per student per assignment. We seek to answer the following research questions:

(1) **Is mutation score a good proxy for manually-seeded fault detection rate?** We examine whether mutation score is correlated with manually-seeded fault detection. If mutation score is a good proxy for manually-seeded fault detection rate, then instructors could avoid the manual effort required to write those faults. Additionally, if we find any manually-seeded faults that are not coupled to at least one mutant, these faults could suggest a way in which existing mutation testing tools can be improved (perhaps a new mutation operator is needed, for example).
(2) **Is mutation score a good proxy for real student fault detection rate?** We examine whether mutation score is correlated with real student fault detection. When evaluating student test suites against any kind of faults, it is important for the faults to be representative of faults that students might realistically encounter in their own implementation source code. If we find student faults that are not coupled to at least one mutant, this could suggest a way to improve mutation testing tools.
(3) **Can mutation testing be used to strengthen instructor-written test suites?** Ideally, instructor-written test suites should be able to detect all possible faults in student implementations. Since student implementations are assumed to be correct if they pass the instructor test suite, mutation testing potentially provides a way of generating faults with which to evaluate the instructor test suite.
(4) **Are mutants of instructor-written implementations a valid substitute for manually-seeded faults for evaluating student test suite quality?** There is evidence in the literature that mutants are a valid substitute for real faults in open source software. However, since we do not know whether manually-seeded faults are representative of real student faults, we cannot assume that mutants are a valid substitute for manually-seeded faults.

In our results, we find a strong correlation between mutation score and manually-seeded fault detection rate for four out of five assignments. We argue that the weak correlation in the fifth assignment is largely due do deficiencies in the manually-seeded faults. In two of the assignments we study, we have both student implementations (some of which contain faults) and student test suites. We find a moderately strong correlation between mutation score and real fault detection rate. Our findings have implications for software testing researchers, educators, and tool builders alike. Through a case study analysis, we find that mutants generated from multiple implementations of the same specification are likely to represent more real faults than those generated from a single implementation. We conclude with a discussion of the implications of our results and how to effectively use mutation testing tools for evaluating student test suites.

2 BACKGROUND

This section introduces mutation testing, mutation scores, and the approach pioneered by Just et al [12] to evaluate the correlation between real fault detection rate and mutation scores.

The goal of mutation testing is to quantify the ability of a test suite to find faults in a program, thus a test suite with a higher mutation score ought to be a better test suite. To do so, a mutation testing framework creates several *mutants* of the program, where each mutant (ideally) represents an injected fault. It then runs the test suite on all mutants. The *mutation score* of the test suite is the fraction of mutants that it is able to distinguish from the original subject program. To construct a single mutant, a mutation-testing framework applies a single *mutation operator*, e.g., deleting a statement, reversing a comparison, or eliminating a branch condition. The set of available operators naturally affects the variety of generated mutants, and we discuss the operators that our tools employ in Section 4.1. Not every mutation represents a real fault: in the general sense, it is unknowable (without manual analysis) whether a mutation results in equivalent behavior to the original program or not.

However, real faults are more complicated than single mutations, so it is not immediately obvious if a test suite's mutation score is correlated with its ability to detect real faults, which is what ultimately matters. Just et al. present a dataset of real-world Java programs with faults and their fixes. They use this dataset to investigate whether each fault is *coupled* to some mutant by a given test suite, where a fault is coupled to a mutant if the test suite that detects the fault also detects the mutant. They find that 73% of real faults are coupled to a generated mutant. For the remaining uncoupled faults, they suggest new mutation operators, and point out limitations of mutation testing. They also establish that the correlation between mutation score and real fault detection rate is stronger than the correlation between statement coverage and real fault detection rate.

3 RELATED WORK

A major topic of software testing research is: how can we automatically evaluate the effectiveness of a test suite? It is now established that test suite coverage is not always strongly correlated with test suite effectiveness [10, 14]. It is possible to combine several coverage criteria to better evaluate test suite effectiveness [27]. Jia and Harman present a survey on mutation testing [11], which Just et al. [12] show is correlated with real-fault detection, even after controlling for coverage. Mutation score is also correlated with defect density [28]. This supports several applications of mutation analysis, including test suite reduction [21–23].

The effectiveness of mutation testing depends on the kinds of mutants generated, and there are several ways to improve the mutant generation process [4, 13]. While traditional mutation testing applies only a single mutation at a time [11], one line of research examines the efficacy of *higher order mutations*, which are generated by applying multiple mutation operations simultaneously [9, 30]. Our methodology of studying mutation testing on student programming assignment solutions provides another interesting source of data to potentially improve the mutant generation process. In particular, by examining productive mutants that are generated on some student implementations (but not on others), it may be possible to design better higher-order mutation operators that *could* have generated those mutants from any implementation. Research into improving mutation testing tools could also make the correlations that we find even stronger.

There is also evidence that mutation analysis helps programmers write better test suites [17]. This paper shows that mutation score is correlated with fault detection in multiple hidden implementations, which includes implementations with deliberate faults (written by instructors), and implementations with accidental faults (written by students' peers). The results of our study may help to increase adoption of mutation testing in educational settings, and it would be interesting future work to study whether exposing students directly to mutation testing results would result in better test suites.

One potential concern when attempting to generalize these prior studies of mutation testing to a classroom setting is that the kinds of code written by students may not be representative of the code written by experienced developers. This concern draws on established evidence from multiple fields, including computer science, that novices do not approach problems in the same way as experts, and thus produce different kinds of solutions [3, 29].

In an educational context, prior work examined the use of code coverage as a feedback mechanism for improving the quality of student-written tests, finding that improvements in coverage do not result in improved fault finding ability [6]. Later work shows that coverage is a poor indicator of student test quality, and instead develops an approach to grading based on mutation testing [2, 19]. Moreover, in an "all-pairs testing" approach, where students test each others' code, no significant correlation arises between the fault-detection rate of a test suite and its code coverage or its mutation score [7, 20].

However, the student test suites used in the studies that reached this conclusion appear to have come from assignment submissions where students received feedback on the coverage of their test suites rather than some fault-detection metric. Moreover, these prior works do not evaluate the suitability of mutants to stand-in for instructor-written faults. In our work, we find that mutation score is correlated with students' ability to find faults that are manually seeded by an instructor. However, there is a moderately strong correlation between mutation score and the the ability to find faults in other students' implementations. The difference in these results could indicate that giving students actionable feedback on their test suites' ability to detect manually-seeded faults does drive them to write higher quality test suites.

Seeded faults and tests can be constructed and used in a number of ways. E.g., it is possible to use other students' submissions as a source of real faults or as the target for mutation testing [24]. Wrenn et al. [31] discusses several flaws with automated assessment of student code, and recommends evaluating student tests with multiple implementations. Our work shows that mutation testing is a scalable way of generating multiple (faulty) implementations and is as effective as having multiple, manually-seeded faulty implementations.

4 METHODS

Our goal in this study is to determine whether mutation score is an accurate indicator of student test suite quality. We analyze data collected from programming assignments in which students were required to submit source code that conforms to a specification and/or test cases that were evaluated against a set of manually-seeded faults. Using off-the-shelf mutation testing tools, we collect mutation scores for the student test suites and look for a correlation between mutation score and manually-seeded fault detection rate. We then examine whether every mutant is coupled to at least one manually-seeded fault by at least one student-written test case.[1]

We also examine whether mutation score is a good indicator in general for the manually-seeded fault detection rate, independent of statement coverage, using methodology from Just et. al [12]. Prior work has shown that statement coverage has a conflating effect on mutation score. That is, test suites that exercise more statements are also likely to detect more mutants. For each manually-seeded fault, we identify pairs of student test suites, $(\tilde{T}_{fail}, \tilde{T}_{pass})$, where $\tilde{T}_{fail}$ detects the fault and $\tilde{T}_{pass}$ does not. For each pair, we compute an adjusted mutation score that only includes mutants present in code that is covered by both $\tilde{T}_{fail}$ and $\tilde{T}_{pass}$. That is, if $\tilde{T}_{fail}$ covers a mutant that $\tilde{T}_{pass}$ does not (or vice-versa), that mutant will not be included in either test suite's mutation score. We then compare the median adjusted mutation score for the populations of $\tilde{T}_{fail}$ and $\tilde{T}_{pass}$ test suites and use the Wilcoxon signed-rank test to determine if the differences in median are statistically significant.

We also investigate whether mutation score is a good proxy for real student fault detection rate. Since fault isolation is a complex problem, we measure real student fault detection as the number of student implementations detected as containing at least one fault. We record the real student fault detection rate of each student test suite and look for a correlation between mutation score and real student fault detection rate. We also use the instructor test suite and differential testing with the instructor implementation as oracles for the actual number of faulty student implementations.

When recording mutation score, manually-seeded fault detection rate, and real student fault detection rate, we take steps to make sure student-written test cases are free of false positives. We define a false positive as a student test case that fails when run against a correct instructor implementation. We discard tests with false positives using the same policy applied by the instructors when the assignments were graded. For some assignments, the entire test suite was rejected if it contained any false positives. For other assignments, only the specific test cases that contained false positives were discarded. Since students received automated feedback on the presence of false positives in their tests (and therefore the impact on their grade), we know that discarding test suites or test cases in

[1]A mutant is coupled to a fault if there is a test case that detects both the mutant and the fault.

the same way as in the original assignment grading process will not be overly aggressive.

4.1 Mutation Testing Tools Used

We use two open-source mutation testing tools in our study: Stryker Mutator [25] version 5.4.1 for assignments written in JavaScript and TypeScript and Mull [5] version 0.14.0 for assignments written in C++. We enabled all mutation operators supported by Stryker for JavaScript (this is the default option) and all non-experimental mutation operators supported by Mull for C++ (using the option `-mutators=cxx_all`). Stryker applies its mutation operators at the AST level and supports a variety of mutation operators including arithmetic and logical operator replacements, conditional expression replacement, and statement deletion. A full list of supported operators can be found on the Stryker website [26]. In contrast, Mull applies its mutation operators at the bytecode level for faster performance and then maps the bytecode modification back to a source code location to present to the human user. While Mull's list of supported mutation operators [15] includes arithmetic and logical operator replacement, it does not support statement deletion or conditional expression replacement to the same extent that Stryker does. Instead, Mull supports a "remove void call" mutation operator that removes a call to a function that returns void and a "replace scalar call" mutation operator that replaces a call to a function that returns a scalar value with the integer literal 42.

4.2 Datasets

We examined assignments from four courses from three different R1 institutions. To address our research questions, we required the following information:

- **RQ1** Student test suite implementations, which were graded using manually-seeded faults, and which could be executed using an off-the-shelf mutation testing tool
- **RQ2** The same as RQ1, plus student implementations of the system under test
- **RQ3** Student implementations of the system under test, and an instructor-written test suite that could be executed using an off-the-shelf mutation testing tool

We answer RQ4 by synthesizing the results to RQ1, RQ2, and RQ3. We selected programming assignments that met these criteria. Table 1 summarizes key information about the programming assignments we collected data from. It was difficult to identify many assignments that satisfied *all* of the criteria, and hence some assignments are used only to address some of the research questions. We examined a total of 2,711 assignment submissions across six programming assignments taken from a total of four courses from three different R1 institutions. We collected only one submission per student for each assignment. Here we briefly summarize each assignment.

OOP Card Game ("Game Card" and "Game Player"). For this assignment, students implemented abstract data types (ADTs) representing a card in a standard deck of 52 playing cards and a player in a card game. Students also wrote test cases for those ADTs and wrote a command-line application simulating a card game using those ADTs. The ADTs interact with each other (e.g., a player holds cards in their hand), but each of the ADTs were evaluated separately from each other when students submitted their source code. Therefore, we will treat the data collected from the "Game Card" and "Game Player" ADTs as two separate datasets in our analyses.

Students were allowed to work alone or with a partner. We collected 785 assignment submissions total (one submission per student/partnership), of which 768 were usable (i.e., we discarded files with compiler errors) for Game Card and 762 were usable for the Game Player portion of the assignment. Students' ADT implementations were evaluated by an instructor-written test suite, and their test cases were evaluated against a set of manually-seeded faults. Students could submit their work to an automated grading system and receive feedback up to three times per day. For their ADT implementation, students received full feedback (exit status and output) on a few minimal, publicly available test cases. For their test cases, students were shown how many manually-seeded faults their tests detected with no additional information about the faults. The assignment was implemented in C++. We collected the following data from the usable student submissions: mutation scores for every student test suite using the Mull [5] mutation testing tool, the number of manually-seeded faults detected by every student test suite, the number of student implementations that contain at least one fault according to the instructor test suite, and the number of student implementations that contain at least one fault according to another student test suite.

Stable Marriage. Students wrote test cases for a set of instructor-written implementations of the classic Gale-Shapley stable marriage algorithm [8] that share a common interface. Students structured their test cases to randomly generate inputs, pass those inputs to an instructor-specified implementation, and then verify whether the return value is a valid solution for that input. Student test cases were evaluated with several correct stable marriage implementations and eight implementations with manually-seeded faults. Students received feedback from an automated grading system on how many faults their tests detected as frequently as they wished. The assignment was implemented in JavaScript. We collected the following data from 485 student submissions (one submission per student): mutation scores for every student test suite using Stryker Mutator [25] and the number of manually-seeded faults detected by every student test suite.

WebApp. Students wrote test cases for an instructor-written implementation of a REST-based web service. Student test cases were evaluated against a set of manually-seeded faults. Students could submit their work to an automated grading system and receive feedback an unlimited number of times. Students were shown how many manually-seeded faults their tests detected with no additional information about the faults. The assignment was implemented in TypeScript. We collected the following data from 93 student submissions: mutation scores for every student test suite using Stryker Mutator [25] and the number of manually-seeded faults detected by every student test suite.

Sorting. Students wrote test cases for a set of instructor-written sorting implementations that share a common interface. The sorting implementations were bubble sort, heap sort, tree sort, quick sort, and merge sort. Student test cases were evaluated against a set of

Table 1: Summary of the programming assignments we collected data from. A "Yes" in the "Has Student Impls" column indicates that students implemented some software artifact conforming to a specification and submitted their source code implementation for grading. A "Yes" in the "Has Student Tests" column indicates that students wrote and submitted test cases for grading. The test cases were graded by being run against a set of manually-seeded faults, and a grade was assigned based on how many manually-seeded faults the test cases detected. All assignments were graded using some sort of automated grading system that would give students immediate feedback on their work. The "# of Submissions/Day" column indicates the number of times that students were allowed to submit their test cases to the automated grading system and still receive feedback. The "Test Case Feedback" column summarizes the feedback that students received on their test cases from the automated grading system. "# of faults detected" indicates that students were shown how many manually-seeded faults their tests detected, but not which specific faults those were. LOC is lines of code (excluding blank lines and comments) of the instructor implementation from which manually-seeded faults were constructed for the assignment.

Assignment	# of Submissions	Has Student Impls	Has Student Tests	# of Submissions/Day	Test Case Feedback	LOC
Game Card	768	Yes	Yes	3	# of faults detected	136
Game Player	762	Yes	Yes	3	# of faults detected	127
Stable Marriage	485		Yes	Unlimited	# of faults detected	79
WebApp	93		Yes	Unlimited	# of faults detected	265
Sorting	90		Yes	5	# of faults detected	190
Restaurants	513	Yes		n/a	n/a	81

manually-seeded faults. Students could submit their work to an automated grading system and receive feedback up to five times per day. Students were shown how many manually-seeded faults their tests detected with no additional information about the faults. The assignment was implemented in TypeScript. We collected the following data from 90 student submissions: mutation scores for every student test suite using Stryker Mutator [25] and the number of manually-seeded faults detected by every student test suite.

Restaurants. Students implemented six methods to process Yelp restaurant reviews in JSON. Their implementations were evaluated by an instructor-written test suite. Students could submit their work to an automated grading system and receive feedback on a subset of the instructor-written test suite unlimited times per day. The assignment was implemented in JavaScript. We collected the following data from 513 student submissions: the number of student implementations that contained at least one fault according to the instructor-written test suite.

5 EVALUATION

We conduct an analysis of the data we collected from these six programming assignments, addressing each of our research questions.

5.1 RQ1: Is mutation score a good proxy for manually-seeded fault detection rate?

We start by examining the relationship between mutation score (number of mutants detected) and manually-seeded fault detection rate on the five programming assignments in which students submitted test suites. Figure 1 shows scatter plots of mutation score vs. number of manually-seeded faults detected. For all but one of these assignments (the Sorting assignment), we see a strong correlation between mutation score and manually-seeded fault detection. We also examined whether every manually-seeded fault is coupled to at least one mutant by at least one student-written test case and did not find any uncoupled manually-seeded faults. This implies that manually-seeded faults and mutants have a similar capacity to measure test suite quality. It may also suggest that requiring students to write test cases with the goal of detecting an undisclosed set of manually seeded faults guides students towards writing test cases that are capable of detecting mutants.

5.1.1 Sorting Project: Qualitative Analysis. Since we only saw a weak correlation between mutation score and manually-seeded fault detection for the "Sorting" project, we investigate what factors may have contributed to this. First, we examine the two outliers with mutation scores significantly higher than all the other submissions. After looking at which mutants these students detected that other students did not and discussing it with the course instructor, it became clear that these students were testing under-specified parts of the assignment. Specifically, the initial version of the assignment did not specify that the TypeScript compiler should be run with strict null-checks enabled, which created ambiguity about whether students were required to test the sorting implementations with `null` and `undefined` inputs. The instructor informed the students that they did not need to write tests using these inputs and updated the sorting implementations under test to include checks for `null` and `undefined`. The extra mutants that these two students detected were simply changes to these added checks, and since students were told that they need not write tests with `null` and `undefined` inputs, we can safely ignore these outliers. With those outliers removed, the Pearson correlation coefficient becomes 0.35.

Next, we examine the manually-seeded faults used to evaluate students' test cases. It seems that the manually-seeded faults were conceived of as trying to represent obscure edge cases rather than a full range of sorting implementation behaviors. Some examples of these faults include: throwing an exception if the input array is of size one, throwing an exception if the input array has a string as its first element, only sorting the even- or odd-indexed elements of the array, and only sorting the first 256 elements of the array. We believe that these faults are not representative of realistic faults that students might encounter in their own code, and this may have

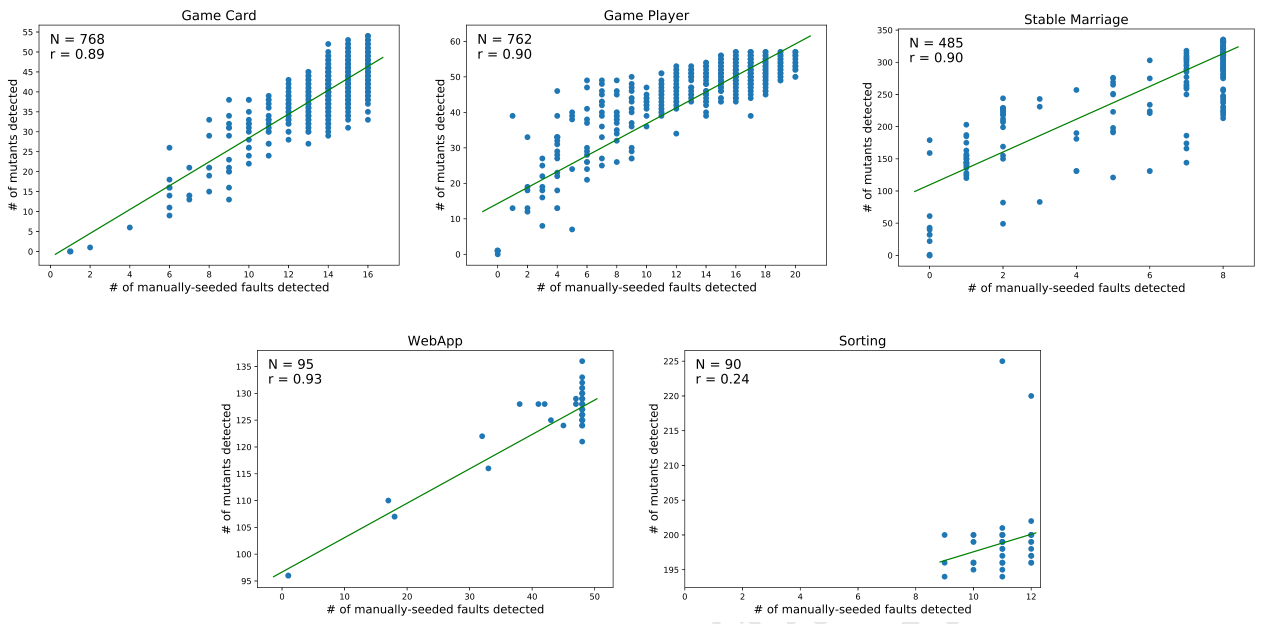

Figure 1: RQ1: Is mutation score a good proxy for manually-seeded fault detection rate? Each dot represents one student-written test suite. The x-axis shows the number of manually-seeded faults detected, and the y-axis shows the number of mutants detected by each student-written test suite. We see a strong correlation for all assignments except for "Sorting."

weakened the correlation between mutation-score and manually-seeded fault detection for this assignment. We discussed this matter with the class' instructional staff, who agreed that these faults did not match the learning objectives for the assignment, and who were interested in following our work to understand if mutation testing could replace the manual fault-seeding process.

5.1.2 Controlling for Coverage. We follow Just et. al's methodology to examine whether a high mutation score is indicative of a high manually-seeded fault detection rate, independent of code coverage. We use $\tilde{T}_{fail}$ to indicate a test suite that detects a particular fault and $\tilde{T}_{pass}$ to indicate a test suite that does *not* detect a particular fault. Taken together, $(\tilde{T}_{fail}, \tilde{T}_{pass})$ indicates a pair of test suites where the first detects a fault and the second does not detect that same fault. For four out of the five projects we analyzed, we see in Table 2 that for every manually-seeded fault for which at least one $(\tilde{T}_{fail}, \tilde{T}_{pass})$ pair exists, the median mutation score of the $\tilde{T}_{fail}$ population is significantly higher than that of the $\tilde{T}_{pass}$ population. For the fifth project (Sorting), we see that this is the case for half of the manually-seeded faults for which at least one $(\tilde{T}_{fail}, \tilde{T}_{pass}$ pair exists).

5.1.3 Sorting Project. For one of the manually-seeded faults, we observe that the $\tilde{T}_{fail}$ population for that fault has a *lower* median mutation score than its corresponding $\tilde{T}_{pass}$ population. The manually-seeded fault in question here is one that throws an exception when the input array is only one element. We suspect that this fault is not representative of a real student fault, and we attempted to write our own more realistic fault that is *only* detectable with an input of size one. That is, the modified implementation should return the wrong answer for inputs of size one but *not for any larger inputs*. We were unable to come up with such a fault after some collective effort. As such, we find it unsurprising that detection of this manually-seeded fault does not imply a higher mutation score.

5.1.4 RQ1 Conclusions. We found no examples of manually-seeded faults that were not coupled to at least one mutant. For four out of five assignments, we see a strong correlation between mutation score and manually-seeded fault detection rate. For the fifth assignment where we saw a weak correlation, we found that many of the manually-seeded faults did not match the learning objectives

Table 2: Summary of $(\tilde{T}_{fail}, \tilde{T}_{pass})$ analysis for each assignment. The middle column states how many of the manually-seeded faults for that assignment had at least one $(\tilde{T}_{fail}, \tilde{T}_{pass})$ pair. The right column states how many of the $\tilde{T}_{fail}$ populations had a significantly higher median mutation score than their corresponding $\tilde{T}_{pass}$ population.

Assignment	**$(\tilde{T}_{fail}, \tilde{T}_{pass})$ populations**	**Significant**
Game Card	15/15	15/15
Game Player	20/20	20/20
Stable Marriage	8/8	8/8
WebApp	47/48	47/47
Sorting	8/12	4/8

of the assignment.[2] We also analyzed whether mutation score is a good indicator in general for manually-seeded fault detection rate, independent of statement coverage, and found in almost every case that detection of a given manually-seeded fault is associated with having a higher mutation score. This evidence supports the conclusion that mutation score is a strong proxy for manually-seeded fault detection rate.

5.2 RQ2: Is mutation score a good proxy for real student fault detection rate?

We next investigate whether mutation score is a good proxy for real student fault detection. As described in Section 4, we define a "real student fault" as a student implementation that failed at least one test, and that this failing test passed when run on the instructor-written implementation. The "real student fault detection rate" for a test suite, then, is the number of faulty student implementations detected by a student-written test suite. We use the Game Card and Game Player assignments in our analysis (note that these are our only two datasets with both student implementations and student-written test suites). Figure 2 shows scatter plots of the number student implementations detected as faulty vs. mutation score for each student test suite for these assignments. We see moderately strong correlations in both cases (0.67 for Game Card and 0.79 for Game Player).

We conduct a case study on the Game Card assignment to determine why the correlation between faulty student implementation detection and mutation score is not as strong as the correlation between manually-seeded fault detection and mutation score. One hypothesis for this weaker correlation is that faults in student code are different than mutants or that student implementations are structured differently from the instructor implementation the mutants were generated from. Another hypothesis is that faults are not evenly distributed throughout the student-written implementations. We investigate these hypotheses by examining faults in student code that the instructor-written test suite did not detect any faults in. If enough of these faults are coupled to a mutant, this would suggest that faults are not evenly distributed throughout student-written implementations. If we find faults that are not coupled to at least one mutant, we then seek to determine whether they might be coupled to a mutant created using a new or modified mutation operator. The key findings of our case study are as follows:

1. Faults are not evenly distributed throughout student implementations. Some student implementations contain more faults than others, and some faults are present in more student implementations than other faults.
2. After adding a few additional test cases to the instructor test suite in order to increase the instructor test suite's mutation score by 10%, we see a 92% increase in the number of student implementations detected as containing at least one fault.
3. Of the remaining undetected faulty student implementations, only 31 (about 5% of all faulty implementations) contain faults that are not coupled to mutants produced from a new or existing mutation operator. The structure of these implementations differs significantly from that of the instructor implementation.

[2] We confirmed this with the TAs and instructors of the course.

5.2.1 *Game Card Assignment Case Study.*

Our case study proceeds as follows: First, we obtain an accurate baseline for the number of student implementations that contain at least one fault. Second, we add test cases to the instructor test suite in order to maximize its mutation score and evaluate which additional student implementations the enhanced instructor test suite detects. Third, we add test cases to the instructor test suite that detect at least one fault in the remaining student implementations and try to determine whether those faults could be coupled to a mutant produced using a new or enhanced mutation operator.

Total Number of Faulty Student Implementations. We obtain three different measurements of the number of student implementations that contain at least one fault: Using the instructor test suite, using the pool of all student test suites, and using differential testing with the instructor implementation. Since the methods students implemented in the Game Card assignment have a relatively small number of legal inputs (there are only 52 playing cards in a standard deck), we were able to conduct exhaustive differential testing on those methods against the instructor-written implementation. That is, for each method in each student implementation, we compared the return value of that student's implementation of that method to the return value of the instructor's implementations of that same method for every legal combination of input arguments. If the student implementation returned a different value from the instructor implementation for any legal input to any of the methods, we marked the student implementation as containing at least one fault. According to the instructor test suite, 250 student implementations contained at least one fault. According to the student test suites combined, 558 student implementations contained at least one fault. According to exhaustive differential testing, 558 student implementations contained at least one fault. This suggests that the instructor-written test suite is not as thorough as it could be, whether by mistake or intentionally.

Maximizing the Instructor Test Suite's Mutation Score. We added additional test cases to the instructor test suite in order to maximize its mutation score. Each new test case was specifically targeted towards one undetected mutant. That is, we wrote each new test with the goal of testing one program behavior. The mutation score of the unmodified instructor test suite was 50/59 (84%). After adding test cases, the mutation score increased to 56/59 (95%). We note that the mutation tool (Mull) actually reported a score of 54/59 (91%) at this point, but we determined that two of the mutants should have been reported as detected. We confirmed this by manually applying those two mutations to the instructor implementation and re-running the instructor test suite on the manually-mutated code. We discuss the limitations of Mull in Section 4.1. We determined that the remaining three mutants were equivalent. The enhanced instructor test suite reported 258 student implementations as containing at least one fault, an increase of about 3%.

During this process, we observed two instances where Mull did not apply mutation operators where we expected it to. The first of these involved replacing a scalar call with 42 (note that in this context this is equivalent to replacing an expression with `true`), and the second involved replacing a less-than operator with less-than-or-equal-to. We suspect that these two mutants were not generated due to limitations in Mull's implementation (bytecode rather than AST manipulation). We generated these two mutants

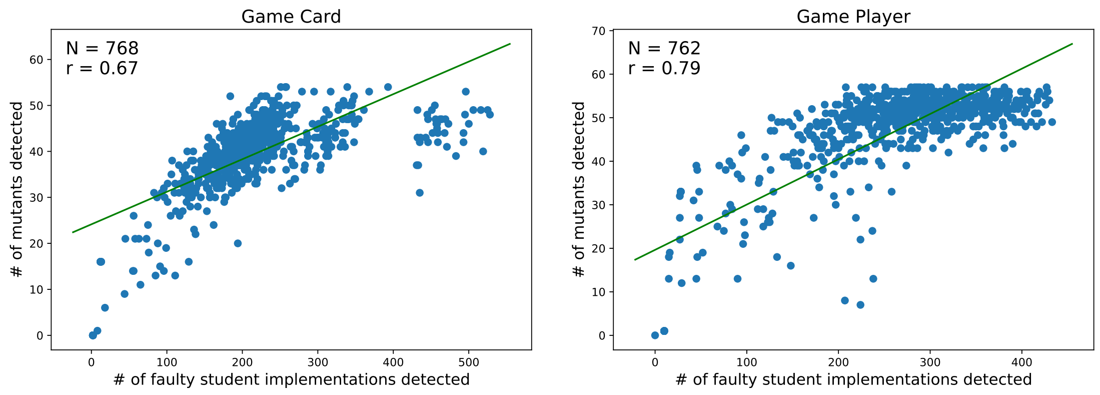

Figure 2: RQ2: Is mutation score a good proxy for real student fault detection rate? Each dot represents one student-written test suite. The x-axis shows the number of student-written implementations detected as having at least one fault, and the y-axis shows the number of mutants detected by each student-written test suite.

by hand and added test cases to the instructor test suite that detect them. This version of the instructor test suite was able to detect 480 faulty student implementations, *a 92% increase over the unmodified instructor test suite* and an 86% increase over the previous version of the enhanced instructor test suite.

New or Modified Mutation Operators. Next, we attempted to determine whether the rest of the undetected faulty student implementations might be coupled to a mutant generated with a new or modified mutation operator. We constructed two more mutants by hand that could be produced with a modified version of the argument omission mutation operator discussed in Just et al [12]. This modified operator would apply argument omission to all identical function calls within a statement instead of only a single call. We added test cases to the instructor test suite that detect these two hypothetical mutants. This version of the instructor test suite then detected 527 faulty student implementations, leaving only 31 more undetected faulty student implementations.

Remaining Undetected Faulty Student Implementations. Of the remaining faulty student implementations, 17/31 (55%) contained at least one unique fault. That is, about half of these faulty implementations were coupled to at least one of the unique faulty implementations. Eight of these 31 faulty student implementations contained faults in their handling of cards with ranks below nine. However, since the card game being implemented does not use these cards, the instructors perhaps did not want to grade students based on those behaviors (note that this behavior *is* well-defined in the specification, but nonetheless represents a corner-case). We focus instead on the 23 other faulty implementations, of which 11 are unique. These faults may not be reproducible as mutants of the instructor implementation because *the structure of those students' implementations differs significantly* from that of the instructor implementation in which the mutants were generated. It may be the case that mutation testing tools could produce mutants coupled to these faults if applied to other correct implementations that are structured differently.

5.2.2 RQ2 Conclusions.

For our pair of assignments that provided both student implementations (some of which were buggy) and student test suites, we found a moderately strong correlation between mutation score and real fault detection rate. We see these moderately strong correlations despite the fact that faults were not evenly distributed throughout student implementations (i.e., some student implementations contain more faults than others, and some faults are present in more student implementations than other faults). After accounting for the limitations of the mutation testing tool we used for those assignments, we found only a small number of student implementations with at least one fault that was not coupled to at least one mutant. We note that the set of mutation operators supported by mutation testing tools is a very important factor in its ability to produce faults that are representative of real student faults. This evidence suggests that mutation score is a reasonably good proxy for real student fault detection rate *even though we only generated mutants from a single instructor-written implementation.* Prior work also supports the notion that generating mutants from a broader range of instructor- or (correct) student-written implementations would strengthen these results [24].

5.3 RQ3: Can mutation testing be used to strengthen instructor-written test suites?

We investigate the extent to which mutation testing can be used to strengthen instructor-written test suites. That is, whether increasing the mutation score of an instructor-written test suite increases its ability to detect real faults in student code. To address this research question, we need only examine student-written solutions of the assignment, and not student-written test cases. We discussed the results of strengthening the instructor test suite for the Game Card assignment in Section 5.2.1. Adding test cases to the instructor suite to maximize its mutation score resulted in a 92% increase in student implementations detected as containing at least one fault.

For the Restaurants assignment, the mutation score of the unmodified instructor test suite was 75/138 (54%). We added five test cases to increase the mutation score to 97/138 (70%), an increase of 29%. Although the new tests did not detect any faults in student implementations that did not already have at least one detected fault, the methods students were required to write do not depend on each other. This allows us to perform some additional analysis. For each added test case, we constructed a list of other test cases in the instructor test suite that might fail because of the same underlying fault. For each of these (`new test case, related old test cases`) pairs, we looked for student implementations that failed the new test case but that did *not* fail any of the related old test cases. After conducting this analysis, we found 217 previously-undetected faults across 179 student implementations (35%). That is, more than a third of the student implementations contained at least one undetected fault. Of these newly-detected faults, five were coupled to a mutant of Method 2, another five were coupled to a mutant of Method 3, 38 were coupled to a mutant of Method 4, 29 were coupled to a mutant of Method 6, and 140 were coupled to another mutant of Method 6.

5.3.1 RQ3 Conclusions. We found two assignments where adding test cases to the instructor-written test suite with the goal of increasing the mutation score resulted in large increases in student fault detection capability. This suggests that mutants are coupled to certain kinds of real student faults that instructors may overlook when writing test suites with which to evaluate student implementations. Additionally, instructors can strengthen their test suites using mutation testing before any students submit implementations and/or test cases with which to evaluate the instructor test suite. Reviewing the output of state-of-the-art mutation testing tools also requires significantly less manual and computational effort than approaches that leverage the pool of student implementations and/or test cases. We recommend that instructors evaluate their own test suites using an off-the-shelf mutation testing tool.

5.4 RQ4: Are mutants a valid substitute for manually-seeded faults for evaluating student test suite quality?

Our results in Section 5.1 show that mutation score is a strong proxy for manually-seeded fault detection rate, and in Section 5.2 we find that mutation score is reasonably good proxy for real student fault detection rate. We found no examples of manually-seeded faults that were not coupled to at least one mutant, and we found relatively few student implementations with at least one fault not coupled to at least one mutant. This evidence supports the conclusion that mutants are a good substitute for manually-seeded faults when evaluating student test suite quality.

6 DISCUSSION

We present the implications of our results for software testing researchers, for software testing educators, and for mutation testing tool builders. We also reflect on the threats to validity of our conclusions and the efforts that we took to mitigate those threats.

6.1 Implications for Researchers

Our work has implications for future work in mutation testing research. Most experiments that have evaluated the suitability of mutation testing to stand in for real faults has considered faults in successive versions of a single implementation of the software under test. However, one of the implicit goals of mutation testing is to measure test suite quality *independent* of implementation structure. Our results suggests a line of work that involves generating mutants from multiple implementations, sourced from student code.

There is a growing trend of using mutation testing in industry, but one of the main priorities in that setting is to reduce the total number of mutants that need to be generated and therefore reduce the computational effort required to run mutation testing tools. Research in which mutants are generated from multiple implementations could help answer the question of which mutants are the most productive for measuring test suite quality.

Although our datasets span several different institutions, there is still a wealth of other instructors who use various strategies to evaluate student test suite quality. Conducting additional research into using test suite quality metrics on student test suites using other datasets could help improve our understanding of the trade-offs of these metrics and strategies. Furthermore, research on the nature of student faults and student test suite quality may help improve our understanding of the differences between novice- and expert-written faults and test suites, which would likely have implications for our understanding of test suite quality metrics.

6.2 Implications for Educators

Our results show that instructors who use manually-seeded faults to evaluate student test suite quality could likely use mutants to generate a broader range of faults (perhaps generating the mutants from multiple implementations). Using an off-the-shelf mutation testing tool requires much less manual effort than writing manually-seeded faults and helps ensure that student tests will be evaluated against realistic faults.

We note that mutation score *should not be directly interpreted as a test suite quality grade* due to equivalent mutants. Either a mutation score threshold for full credit can be applied, or the mutants can be generated ahead of time and equivalent mutants discarded. Generating the mutants ahead of time has the added benefit of reducing the computational overhead of mutants that result in timeouts during grading. There is some discussion of this process in prior work [19]. Additionally, instructors can use mutation testing on their own test suites in order to help ensure that these instructor-written test suites exercise a broad range of program behaviors in student implementations. We believe that our study will provide educators with additional confidence to use mutation testing to grade student test suites, and that these collective experiences will help to better determine how to use the results in grading.

In our experiments, we evaluated student test suites against mutants generated from the instructor's reference implementation. Prior work explored evaluating student test suites against mutants generated from the same student's implementation [2]. While using this approach for grading has notable drawbacks, it may be worth revisiting the question of whether students should be encouraged to use mutation testing on their own implementations, outside of the assignment submission feedback loop. Prior work suggests that students benefit from frequent, actionable feedback [6, 24], and teaching students how to apply mutation testing on their own may give them additional opportunities to receive feedback on their work. This may help improve students' ability to reason about their source code through the process of determining whether undetected mutants are equivalent. Code Defenders [18] is an interesting example of how these learning goals can be combined with gamification, and perhaps there is future work that could explore the use of mutation testing tools in such a context.

Finally, our findings suggest that mutation testing tools have untapped potential in educational settings, and we look forward to engaging with the community on this topic. We plan to release publicly-available information about the assignments we used in our evaluation so that other instructors can use them as a reference for how to structure future assignments that involve evaluating student test suite quality.

6.3 Implications for Tool Builders

Tool builders may be interested in providing better support for educational applications of mutation testing, since ease of adoption for instructors may improve the visibility of those tools. Mutation testing tools are often designed for the use case where the tool is run once, the results analyzed, the test suite improved, and then the tool is re-run, and results re-analyzed. The output of these tools is typically an HTML report that shows mutants that were and were not detected, as well as overall summary statistics.

To effectively apply mutation testing to grade student test suites, it is most useful for the mutation testing tool to support a distinct "mutant generation" phase, where an instructor can determine which mutants should be executed on subsequent executions of student code. Similarly, it is necessary for the mutation testing tool to provide some stable mapping of the mutations that are detected by a test suite, so that it is possible to determine which mutants were detected by an instructor test suite, but not a student test suite. Stryker and Mull both support reporting their results as a JSON file that follows the Mutation Testing Report schema [1], which makes it possible to develop portable utility programs that could provide initial support for these features. While being able to independently develop such utility programs is a useful feature, educators should work with mutation tool builders to standardize these interfaces and integrate such features into the tools themselves, which could make it easier to adopt the tools in class.

Our results suggest potential use cases for more easily comparing the mutation scores of multiple test suites, generating mutants from multiple implementations, and pre-generating mutants. Tool builders could also support features that help measure mutant productivity (i.e., which mutants are more likely to illicit an effective test [13]). For example, mutation testing tools could support comparing the mutation scores of multiple test suites so that software developers could examine how a test suite evolves over time.

6.4 Threats to Validity

Construct: Are we asking the right questions? Our research questions are based on established research questions from the mutation testing literature. We posed our new research questions before we examined our dataset. These questions were prompted by our experience developing instructor-written faults and test suites, and anecdotal evidence that they can be inadequate.

Internal: Do our methods and datasets affect the accuracy of our results? Many of our research questions require assignments where student-written tests are graded by their ability to detect instructor-written faults. When evaluating the relationship between mutation score and real student fault detection, we were only able to include two assignments from the same course, as the other assignments required students to submit only their test suites and not their source implementations. Our case study only examined one of these two assignments, as sifting through faults in hundreds of student implementations detected by hundreds of student test suites requires a tremendous amount of manual effort. We were only able to record the number of student implementations containing at least one fault rather than the total number of real faults. Fault localization is a challenging problem with its own body of research, and manual fault localization for this many submissions is impractical.

There could be bugs in the scripts that we wrote, or the tools that we used. We carefully examined the output of each step in our analysis, and investigated discrepancies. We observed a few concerning behaviors when using Mull, although we did not find these deficiencies to make a significant impact on our final conclusions. In some cases, Mull did not apply its mutation operators in places we expected it to. We also observed two instances where Mull reported a particular mutant as undetected even though we independently verified that the test suite in question did actually detect that mutant. We manually applied the mutation to a copy of the implementation source code, ran the test suite, and noted several test cases failing, which suggests potential bugs in Mull.

In order to determine the severity of this discrepancy, we conducted an experiment on a single assignment (Game Card), where we manually seeded all of the mutants that Mull reported to have created. The results of this experiment were sufficiently similar to the results that we reported in Section 5.2 that we determined that any discrepancies caused by Mull's implementation decisions do not influence our conclusions. Hence, despite the possibility for bugs in this tool, we feel confident that our analysis of the mutants, test results, and results hold.

External: Would our results generalize? Our evaluation uses six programming assignments, and they may not be representative of every kind of programming assignment. However, our assignments were drawn from four different courses, from three different institutions, and have 2,711 total submissions. The assignments are in two programming languages and use two different off-the-shelf mutation testing tools. While we are not permitted to release student submissions, we will make our entire evaluation and analysis pipeline publicly available so that other researchers may replicate our work using a different set of student submissions.

It is not the case that mutation testing can help strengthen every instructor-written test suite. We investigated instructor-written test suites from three other assignments, and found that those test suites were already extremely strong, and could not be improved further using mutation analysis or manual inspection. We examined every mutant that was not detected, and verified that they were equivalent to the instructor-written solution.

7 CONCLUSION

We investigated whether mutants can be used in place of manually-seeded faults when evaluating student test suite quality. Our results show that the open-source mutation testing tools we used in our evaluation produce mutants of equal or higher quality than manually-seeded faults written by instructors on all five programming assignments we evaluated. We recommend that instructors use mutants instead of manually-seeded faults when evaluating student test suite quality, as writing manually-seeded faults can be error-prone. We also found that mutants generated from a single instructor-written implementation are a reasonably good stand-in for real faults in student implementations. Generating mutants from additional implementations that are structured differently would likely yield even better results. Future research in mutation testing should consider evaluating mutants generated from multiple implementations of the same system under test when feasible.

REFERENCES

[1] 2022. Mutation Testing Report Schema. https://github.com/stryker-mutator/mutation-testing-elements/tree/master/packages/report-schema

[2] Kalle Aaltonen, Petri Ihantola, and Otto Seppälä. 2010. Mutation Analysis vs. Code Coverage in Automated Assessment of Students' Testing Skills. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion* (Reno/Tahoe, Nevada, USA) *(OOPSLA '10)*. Association for Computing Machinery, New York, NY, USA, 153–160. https://doi.org/10.1145/1869542.1869567

[3] Michelene T. H. Chi, Robert Glaser, and Ernest Rees. 1982. Expertise in problem solving. *Advances in the psychology of human intelligence* Vol. 1 (1982), 7–76.

[4] Pedro Delgado-Pérez, Louis M. Rose, and Inmaculada Medina-Bulo. 2019. Coverage-Based Quality Metric of Mutation Operators for Test Suite Improvement. *Software Quality Journal* 27, 2 (jun 2019), 823–859.

[5] A. Denisov and S. Pankevich. 2018. Mull It Over: Mutation Testing Based on LLVM. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 25–31. https://doi.org/10.1109/ICSTW.2018.00024

[6] Stephen H. Edwards. 2003. Improving Student Performance by Evaluating How Well Students Test Their Own Programs. *J. Educ. Resour. Comput.* 3, 3 (sep 2003), 1–es. https://doi.org/10.1145/1029994.1029995

[7] Stephen H. Edwards and Zalia Shams. 2014. Comparing Test Quality Measures for Assessing Student-Written Tests. In *Companion Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE Companion 2014)*. Association for Computing Machinery, New York, NY, USA, 354–363. https://doi.org/10.1145/2591062.2591164

[8] D. Gale and L. S. Shapley. 1962. College Admissions and the Stability of Marriage. *The American Mathematical Monthly* 69, 1 (1962), 9–15. http://www.jstor.org/stable/2312726

[9] Mark Harman, Yue Jia, Pedro Reales Mateo, and Macario Polo. 2014. Angels and Monsters: An Empirical Investigation of Potential Test Effectiveness and Efficiency Improvement from Strongly Subsuming Higher Order Mutation. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. Association for Computing Machinery, New York, NY, USA, 397–408. https://doi.org/10.1145/2642937.2643008

[10] Laura Inozemtseva and Reid Holmes. 2014. Coverage is Not Strongly Correlated with Test Suite Effectiveness. In *International Conference on Software Engineering (ICSE)*.

[11] Yue Jia and Mark Harman. 2011. An analysis and survey of the development of mutation testing. *TSE* 37, 5 (2011).

[12] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are Mutants a Valid Substitute for Real Faults in Software Testing?. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 654–665. https://doi.org/10.1145/2635868.2635929

[13] René Just, Bob Kurtz, and Paul Ammann. 2017. Inferring Mutant Utility from Program Context. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 284–294. https://doi.org/10.1145/3092703.3092732

[14] Pavneet Singh Kochhar, Ferdian Thung, and David Lo. 2015. Code coverage and test suite effectiveness: Empirical study with real bugs in large systems. In *International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 560–564. https://doi.org/10.1109/SANER.2015.7081877

[15] Mull Supported Mutation Operators 2021. Mull: Supported Mutation Operators. https://mull.readthedocs.io/en/0.14.0/SupportedMutations.html

[16] Goran Petrovic. 2021. Mutation Testing. https://testing.googleblog.com/2021/04/mutation-testing.html

[17] Goran Petrović, Marko Ivanković, Gordon Fraser, and René Just. 2021. Does mutation testing improve testing practices?. In *Proceedings of the International Conference on Software Engineering (ICSE'21)*.

[18] José Miguel Rojas and Gordon Fraser. 2016. Code Defenders: A Mutation Testing Game. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 162–167. https://doi.org/10.1109/ICSTW.2016.43

[19] Zalia Shams and Stephen H. Edwards. 2013. Toward Practical Mutation Analysis for Evaluating the Quality of Student-Written Software Tests. In *Proceedings of the Ninth Annual International ACM Conference on International Computing Education Research* (San Diego, San California, USA) *(ICER '13)*. Association for Computing Machinery, New York, NY, USA, 53–58. https://doi.org/10.1145/2493394.2493402

[20] Zalia Shams and Stephen H. Edwards. 2015. Checked Coverage and Object Branch Coverage: New Alternatives for Assessing Student-Written Tests. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (Kansas City, Missouri, USA) *(SIGCSE '15)*. Association for Computing Machinery, New York, NY, USA, 534–539. https://doi.org/10.1145/2676723.2677300

[21] August Shi, Alex Gyori, Milos Gligoric, Andrey Zaytsev, and Darko Marinov. 2014. Balancing Trade-Offs in Test-Suite Reduction. In *Proceedings of the 22nd*

ACM SIGSOFT International Symposium on Foundations of Software Engineering (Hong Kong, China) *(FSE 2014)*. Association for Computing Machinery, New York, NY, USA, 246–256. https://doi.org/10.1145/2635868.2635921

[22] August Shi, Alex Gyori, Suleman Mahmood, Peiyuan Zhao, and Darko Marinov. 2018. Evaluating test-suite reduction in real software evolution. In *ISSTA*.

[23] August Shi, Tifany Yung, Alex Gyori, and Darko Marinov. 2015. Comparing and combining test-suite reduction and regression test selection. In *ESEC/FSE*.

[24] Rebecca Smith, Terry Tang, Joe Warren, and Scott Rixner. 2017. An Automated System for Interactively Learning Software Testing. In *Proceedings of the 2017 ACM Conference on Innovation and Technology in Computer Science Education* (Bologna, Italy) *(ITiCSE '17)*. Association for Computing Machinery, New York, NY, USA, 98–103. https://doi.org/10.1145/3059009.3059022

[25] Stryker 2022. Stryker Mutator. https://stryker-mutator.io/

[26] Stryker Supported Mutators 2022. Stryker Supported Mutators. https://stryker-mutator.io/docs/mutation-testing-elements/supported-mutators/

[27] Dávid Tengeri, Árpád Beszédes, Tamás Gergely, László Vidács, Dávid Havas, and Tibor Gyimóthy. 2015. Beyond code coverage - An approach for test suite assessment and improvement. In *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*.

[28] Dávid Tengeri, László Vidács, Árpád Beszédes, Judit Jász, Gergõ Balogh, Béla Vancsics, and Tibor Gyimóthy. 2016. Relating Code Coverage, Mutation Score and Test Suite Reducibility to Defect Density. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. 174–179. https://doi.org/10.1109/ICSTW.2016.25

[29] Mark Weiser and Joan Shertz. 1983. Programming problem representation in novice and expert programmers. *International Journal of Man-Machine Studies* 19, 4 (1983), 391–398. https://doi.org/10.1016/S0020-7373(83)80061-3

[30] Chu-Pan Wong, Jens Meinicke, Leo Chen, João P. Diniz, Christian Kästner, and Eduardo Figueiredo. 2020. *Efficiently Finding Higher-Order Mutants*. Association for Computing Machinery, New York, NY, USA, 1165–1177. https://doi.org/10.1145/3368089.3409713

[31] John Wrenn, Shriram Krishnamurthi, and Kathi Fisler. 2018. Who Tests the Testers?. In *Proceedings of the 2018 ACM Conference on International Computing Education Research* (Espoo, Finland) *(ICER '18)*. Association for Computing Machinery, New York, NY, USA, 51–59. https://doi.org/10.1145/3230977.3230999