On the Use of Mutation Analysis for Evaluating Student Test Suite Quality

James Perretta[1], Andrew DeOrio[2], Arjun Guha[1], 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

A common practice in computer science courses is to evaluate student-written test suites against either a set of manually-seeded faults (handwritten by an instructor) or against all other student-written implementations ("all-pairs" grading). However, manually seeding faults is a time consuming and potentially error-prone process, and the all-pairs approach requires significant manual and computational effort to apply fairly and accurately. Mutation analysis, which automatically seeds potential faults in an implementation, is a possible alternative to these test suite evaluation approaches. Although there is evidence in the literature that mutants are a valid substitute for real faults in large open-source software projects, it is unclear whether mutants are representative of the kinds of faults that students make. If mutants are a valid substitute for faults found in student-written code, and if mutant detection is correlated with manually-seeded fault detection and faulty student implementation detection, then instructors can instead evaluate student test suites using mutants generated by open-source mutation analysis tools.

Using a dataset of 2,711 student assignment submissions, we empirically evaluate whether mutation score is a good proxy for manually-seeded fault detection rate and faulty student implementation detection rate. Our results show a strong correlation between mutation score and manually-seeded fault detection rate and a moderately strong correlation between mutation score and faulty student implementation detection. We identify a handful of faults in student implementations that, to be coupled to a mutant, would require new or stronger mutation operators or applying mutation operators to an implementation with a different structure than the instructor-written implementation. We also find that this correlation is limited by the fact that faults are not distributed evenly throughout student code, a known drawback of all-pairs grading. Our results suggest that mutants produced by open-source mutation analysis tools are of equal or higher quality than manually-seeded faults and a reasonably good stand-in for real faults in student implementations. Our findings have implications for software testing researchers, educators, and tool builders alike.

ISSTA '22, July 18–22, 2022, Virtual, South Korea

ACM ISBN 978-1-4503-9379-9/22/07…$15.00
https://doi.org/10.1145/3533767.3534217

CCS CONCEPTS

• **Software and its engineering → Software defect analysis**; **Software maintenance tools**.

KEYWORDS

mutation analysis, software testing, software faults

ACM Reference Format:
James Perretta, Andrew DeOrio, Arjun Guha and Jonathan Bell. 2022. On the Use of Mutation Analysis for Evaluating Student Test Suite Quality. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3533767.3534217

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 [8]. 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, 25]. Some instructors use an "all-pairs" approach where every student-written test suite is run against every other student-written implementation [9]. 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 [40]. 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 faults are detected by each student-written test suite [40]. This strategy gives the instructor full control over the number and type of faults used to evaluate student test suites, but requires significant effort. Moreover, instructor-seeded faults may not be representative of all student faults, as students tend to approach problems in fundamentally different ways than experts [3, 38].

An alternative to these grading approaches is *mutation analysis*, a practice that is gaining adoption in industry [20] and open-source

software projects [5, 34]. Prior work has explored using mutation analysis to evaluate student-written test suites [2, 24, 29], but there is no evidence that mutation analysis 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 [14], but it is unclear whether these results also apply to student-written code. One of the fundamental assumptions behind mutation analysis is that the source code under test is usually correct or near-correct (the competent programmer hypothesis) [33], but this assumption may not hold if student-written implementations contain fundamental flaws in their approach. Making matters more confusing, prior work has reached conflicting conclusions on the question of whether mutation analysis is an effective means of evaluating student test suite quality [9, 25, 29].

Contributions: In this paper, we examine the question: "is mutation analysis an effective means of evaluating student test suites?" We conduct a large-scale empirical evaluation of student test suites in two assumed grading scenarios: one where student test suites are evaluated against a set of manually-seeded faults written by an instructor, and one where student test suites are evaluated against all other student-written implementations ("all-pairs"). 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 dataset includes a total of 2,711 assignment submissions across five programming assignments from a total of three courses at three different 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 rate. 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 manually-seeded faults that are not coupled to any mutant, these faults could suggest a way in which existing mutation analysis tools can be improved (perhaps a new mutation operator is needed, for example).
(2) **Is mutation score a good proxy for faulty student implementation detection rate in an "all-pairs" grading approach?** We examine whether mutation score is correlated with faulty student implementation detection rate as measured in an "all-pairs" grading approach. That is, whether mutation score is correlated with the number of student-written implementations detected as containing at least one fault. If we find faults in student-written implementations that are not coupled to any mutant, this could suggest a way to improve mutation analysis tools.

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. On the two assignments for which we have both student implementations and student test suites, we find a moderately strong correlation between mutation score and faulty student implementation 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 analysis tools for evaluating student test suites.

2 BACKGROUND

This section introduces our two assumed grading scenarios (manually-seeded faults and all-pairs grading), mutation analysis, mutation scores, and the approach pioneered by Just et al. [14] to evaluate the correlation between real fault detection rate and mutation scores.

Mutation Analysis. The goal of mutation analysis 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 analysis 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 analysis 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 3.1. Not every mutation represents a real fault: in the general sense, it is unknowable (without manual analysis) whether a mutant is equivalent to the original program or not.

However, real faults are more complicated than single mutations, so it is not immediately clear that a test suite's mutation score is correlated with its ability to detect real faults. 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 analysis. 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.

Grading Scenario 1: (Manually-seeded faults). In this grading scenario, an instructor manually introduces faults into their own correct implementation of an assignment to produce a set of implementations containing one fault each. Students are awarded points based on how many manually-seeded faults their test cases detect. In typical usage, the student test suites are first run against a correct implementation in order to detect false positives (otherwise a student could trivially achieve full credit with a single test case

that always fails). An advantage of this approach is that it gives the instructor total control over the type and number of faults that student test suites are evaluated against. However, this manual approach is time-consuming and runs the risk of omitting important faults or introducing faults that are too difficult to detect. Our study investigates whether mutants are a valid substitute for these manually-seeded faults.

Grading Scenario 2: "All-pairs". Some instructors utilize the collection of all student-written test suites and implementations in order to evaluate both the correctness of the implementations and the quality of the test suites [40]. This approach is commonly referred to as "all-pairs" grading. Students lose points for implementation correctness if another student's test suite reports failures when run with their own implementation, and students are awarded points for test suite quality for each other student-written implementation that their test suite detects as containing at least one fault. Since our study is concerned with evaluating student-written test suites, we focus only on the latter aspect of all-pairs grading (evaluating student-written test suites for their ability to detect faulty student implementations). While this approach has the benefit of evaluating student test suites against real faults in student-written programs, it is difficult to apply accurately [40]. For example, controlling for false positives in student-written test suites (i.e., weeding out test cases that incorrectly report faults in correct implementations) is especially important, otherwise some students would be unfairly rewarded for detecting more faulty implementations than their test suite actually should. Additionally, the number of *(implementation, test suite)* pairs scales super-linearly with the number of students, making this process difficult to scale. Our study investigates the extent to which mutation analysis can be used as a substitute for all-pairs test suite grading.

3 METHODS

Our goal in this study is to determine whether mutation score is an accurate indicator of student test suite quality. We consider two grading scenarios for evaluating student test suite quality: one in which student-written test cases are evaluated against a set of manually-seeded faults (written by an instructor), and one in which each student-written test suite is run against every other student-written implementation ("all-pairs").

Grading Scenario 1 (Manually-seeded faults). In this grading scenario, students are awarded points based on how many manually-seeded faults their test cases detect. We analyze data collected from programming assignments in which students were required to submit test cases that were evaluated against a set of manually-seeded faults. Using off-the-shelf mutation analysis 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.

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 [14]. 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.

Grading Scenario 2 ("All-pairs"). In this grading scenario, students are awarded points based on how many student-written implementations their test cases detect as faulty. An implementation is considered faulty if at least one test case fails when run against it. We analyze data collected from programming assignments in which students were required to submit source code that conforms to a specification and test cases that were evaluated against a set of manually-seeded faults. Using off-the-shelf mutation analysis tools, we collect mutation scores for the student test suites and look for a correlation between mutation score and faulty student implementation detection rate. We then look for student-written implementations that contain faults that are not coupled to a mutant by the instructor-written test suite using the following process:

(1) If the instructor-written test suite does not have the maximum possible mutation score, we add targeted test cases to the instructor-written test suite that increase its mutation score as much as possible.
(2) We obtain a baseline for the set of faulty student implementations using a combination of differential testing and the pool of all student-written test suites.
(3) We examine faulty student implementations undetected by the max-mutation-score version of the instructor-written test suite and determine whether new or stronger mutation operators could generate mutants coupled to these faults.

We also account for the confounding factor that faults may not be evenly distributed throughout student-written implementations. That is, equivalent faults may be present in more than one implementation, and each implementation may contain multiple faults. We seek to determine the extent to which the uneven distribution of the not-mutant-coupled faults we discover contribute to this effect. For each unique not-mutant-coupled fault, we construct a list of student-written implementations that contain at least one not-mutant-coupled fault and *no other faults*. Then for each student-written test suite, we update its set of detected faulty student implementations by subtracting out the faulty student-written implementations that contain only not-mutant-coupled faults. We then recompute the correlation between mutation score and faulty student implementation detection rate using these updated sets. If the recomputed correlation is stronger than the original, then the original correlation is likely weaker because of the uneven distribution of faults that are not coupled to a mutant.

Guarding Against 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 specific test case containing the false positive was discarded, while in others, the entire test suite containing those test case(s) was 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 using the same policy as the original assignment grading process will not be overly aggressive.

3.1 Mutation Analysis Tools Used

We use two open-source mutation analysis tools in our study: Stryker Mutator [34] version 5.4.1 for assignments written in JavaScript and TypeScript and Mull [7] 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 [35]. In contrast, Mull applies its mutation operators at the LLVM 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 [17] 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.

3.2 Datasets

We examined assignments from a total of three courses from the University of Michigan; University of Massachusetts, Amherst; and Northeastern University. To address our research questions, we required the following information:

- **RQ1** Student test suites, which were graded using manually-seeded faults, and which could be executed using an off-the-shelf mutation analysis tool.
- **RQ2** The same as RQ1, plus student implementations of the system under test. We use these student-written test suites and implementations to simulate test suite evaluation in an "all-pairs" grading scenario.

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 five programming assignments. 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 Euchre, a trick-taking card game [19]. Students also wrote test cases for those ADTs and wrote a command-line application simulating a game of Euchre 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 for Game Card and 762 were usable for the Game Player portion of the assignment (i.e., we discarded files with compiler errors). 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 [7] mutation analysis tool, the set of manually-seeded faults detected by every student test suite, the set of student implementations that contain at least one fault according to the instructor test suite, and the set 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 [10] that shared a common interface [19]. Students structured their test cases to randomly generate inputs, pass those inputs to an instructor-specified implementation, and then determine 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 [34] 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 [19]. 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 [34] and the number of manually-seeded faults detected by every student test suite.

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. Students wrote and submitted test cases for grading for all assignments. All assignments were graded using some sort of automated grading system that would give students immediate feedback on their work, indicating the number of manually-seeded faults detected. 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. 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	School	Course	# of Submissions	Has Student Impls	# of Submissions/Day	LOC
Game Card [19]	UMich	EECS 280 [32]	768	Yes	3	136
Game Player [19]	UMich	EECS 280 [32]	762	Yes	3	127
Stable Marriage [19]	UMass	CS220 [30]	485		Unlimited	79
WebApp [19]	Northeastern	CS4530 [31]	93		Unlimited	265
Sorting [19]	Northeastern	CS4530 [31]	90		5	190

Sorting. Students wrote test cases for a set of instructor-written sorting implementations that share a common interface [19]. The sorting implementations were bubble sort, heap sort, tree sort, quick sort, and merge sort. 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 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 [34] and the number of manually-seeded faults detected by every student test suite.

4 EVALUATION

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

4.1 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.

4.1.1 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 mutations to these 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 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 analysis could replace the manual fault-seeding process.

4.1.2 Controlling for Coverage. We follow Just et. al's methodology to examine if 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

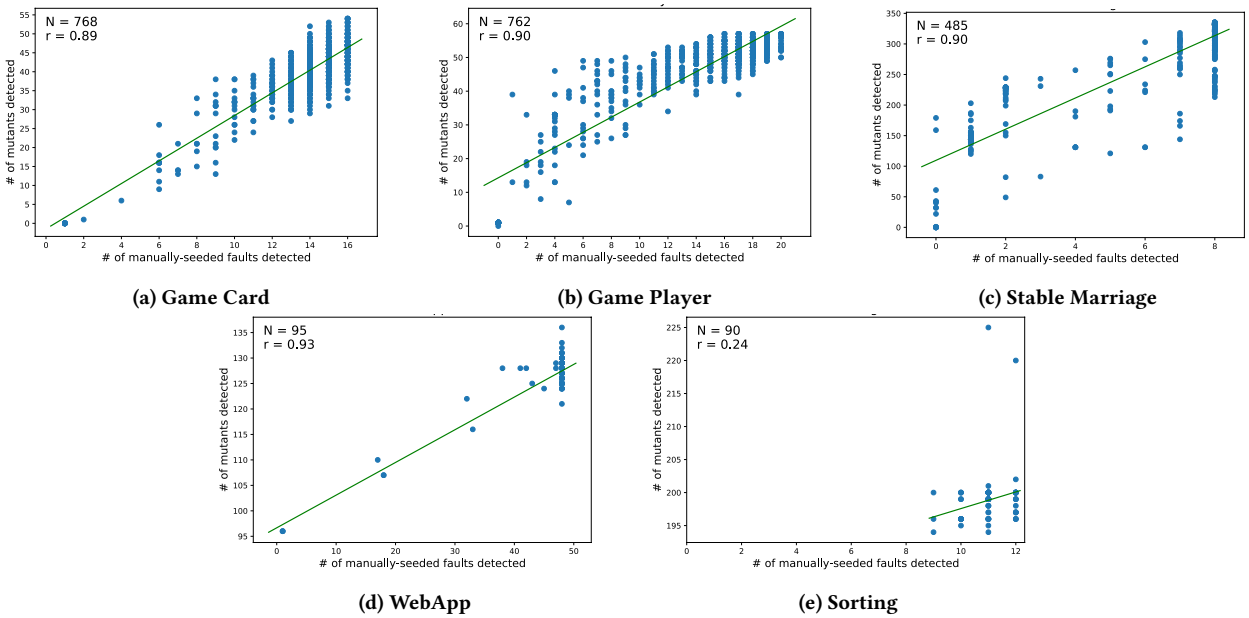


Figure 1: RQ1: Is mutation score a good proxy for manually-seeded fault detection rate? Each dot represents one student-written test suite. The x-axis shows the number of manually-seeded faults detected, and the y-axis shows the number of mutants detected by each student-written test suite. We see a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

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).

4.1.3 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.

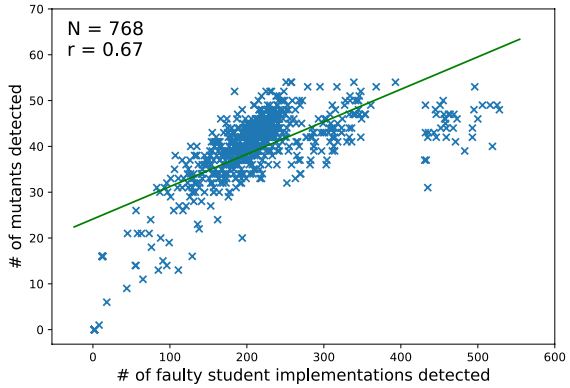
Table 2: Summary of ($\tilde{T}_{fail}$, $\tilde{T}_{pass}$) analysis for each assignment. We show how many manually-seeded faults had at least one ($\tilde{T}_{fail}$, $\tilde{T}_{pass}$) pair. The right column states how many of the $\tilde{T}_{fail}$ populations had a significantly higher median mutation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

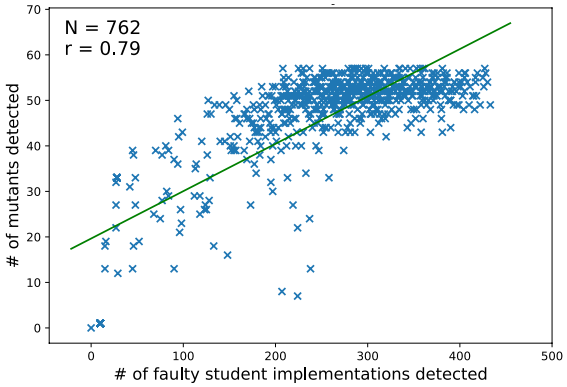
4.1.4 RQ1 Conclusions. We found no examples of manually-seeded faults that were not coupled to at least one mutant. For four out of five assignments, we see a strong correlation between mutation score and manually-seeded fault detection rate. We also analyzed whether mutation score is a good indicator 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.

4.2 RQ2: Is mutation score a good proxy for faulty student implementation detection rate in an "all-pairs" grading approach?

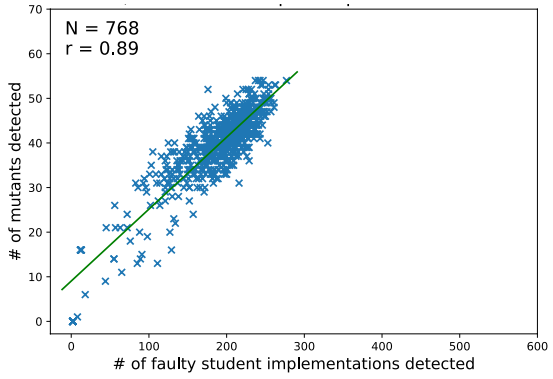
We next investigate whether mutation score is a good proxy for faulty student implementation detection in an all-pairs grading scenario. In this grading scenario, students are awarded points based on how many student-written implementations their test cases detect as faulty. An implementation is considered faulty if at least one test case fails when run against it. 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 of student implementations detected as faulty vs. mutation score for each student test suite for these assignments. We see



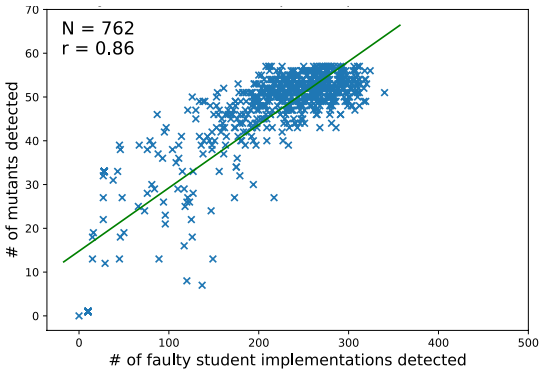
(a) Game Card, considering all student implementations



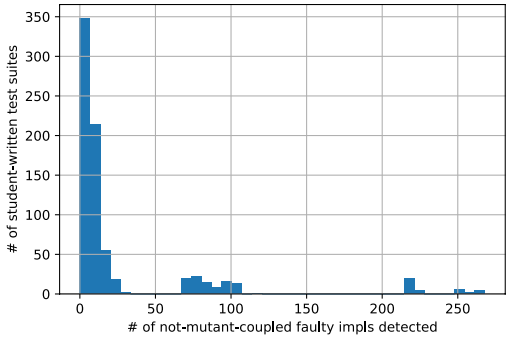
(b) Game Player, considering all student implementations



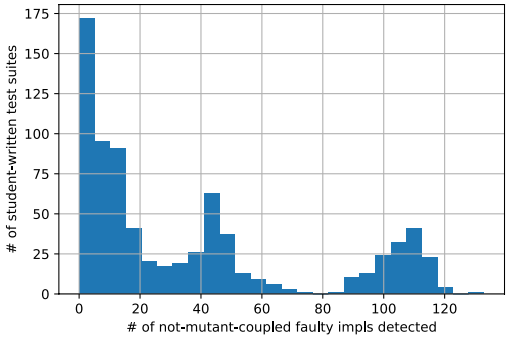
(c) Game Card, implementations with mutant-coupled faults



(d) Game Player, implementations with mutant-coupled faults



(e) Game Card



(f) Game Player

Figure 2: RQ2: Is mutation score a good proxy for faulty student implementation detection rate? Figures 2a and 2b show the relationship between mutant detection and faulty student implementation detection. We see moderately strong correlations between mutant detection and faulty student implementation detection in both assignments (0.67 for Game Card and 0.79 for Game Player). 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. Figures 2c and 2d show the relationship between mutation detection and faulty student implementation detection with not-mutant-coupled implementations removed. That is, we subtracted out faulty implementations that contained no faults besides those that are not coupled to a mutant. With this adjustment, we see the correlation between mutant detection and faulty student implementation detection increase from 0.67 to 0.89 in the Game Card assignment and from 0.79 to 0.86 in the Game Player assignment. Figures 2e and 2f show the distribution of not-mutant-coupled faulty implementations detected by student-written test suites. We observe that the bumps in Figure 2e (around 65-105 and 210-260) seem to align with the two non-linear groups in Figure 2a, suggesting that this non-linearity was caused by a small handful of not-mutant-coupled faults that were present in a large number of student-written implementations. Although this effect is somewhat less apparent in Figure 2f, we observe two peaks (around 40-50 and 100-120) that likely contributed to the long tail in Figure 2b. In addition, we see in Figure 2f that a handful of test suites with low mutation scores (25 mutants detected or fewer) detected 100 fewer faulty implementations after removing the not-mutant-coupled faulty implementations.

moderately strong correlations in both cases (0.67 for Game Card and 0.79 for Game Player).

We found 15 unique not-mutant-coupled faults in student implementations of Game Card and 26 unique not-mutant-coupled faults in student implementations of Game Player. Three of the Game Card faults would require a new or stronger mutation operator, five require an existing mutation operator not supported by Mull, and seven would require a mutant generated from an implementation with a different structure (we did not find any Game Card faults that are not coupled to a mutant). Two of the Game Player faults would require a new or stronger mutation operator, nine require an existing mutation operator not supported by Mull, one would require a mutant generated from an implementation with a different structure, and 14 are not coupled to any mutant. We note that while eight of these last 14 Game Player faults have the same root cause, we count them separately because they are detected by different test inputs. We describe these faults in more detail in Sections 4.2.1 and 4.2.2.

We also examined the impact of these not-mutant-coupled faults and their potentially uneven distribution throughout student implementations. After removing these faulty implementations from our datasets, we see the correlation between mutation score and faulty student implementation rate increase from 0.67 to 0.89 for Game Card and from 0.79 to 0.86 for Game Player. Figures 2c through 2f contain scatter plots for these updated datasets and histograms of how many student implementations contained only not-mutant-coupled faults. We note that we uncovered a relatively small number of unique faults not coupled to a mutant in our investigation (15 unique faults in Game Card and 26 unique faults in Game Player) despite examining hundreds of faulty student implementations. This suggests that the impact of these faults on our initial results was exacerbated by the fact that the faults are not evenly distributed throughout student-written implementations, which is a known drawback of the all-pairs grading approach.

4.2.1 Not-Mutant-Coupled Game Card Faults. We found 15 unique faults in student-written Game Card implementations that are not coupled to a mutant generated by Mull from the instructor-written implementation. All of these faults were found in one of two functions that compare cards while taking into account the trump suit and/or the suit of the led card for the current trick. Although these functions can be implemented with less than 10 conditional cases each, some student-written implementations are much more complex, which results in new source code locations where faults can be introduced.

Following the process described in Section 3, we first conducted mutation analysis on the instructor-written test suite. Unmodified, the instructor test suite had a mutation score of 50/59 (84%). After adding test cases targeted specifically towards those undetected mutants, the mutation score increased to 56/59 (95%). We note that two mutants were originally reported by the mutation analysis tool (Mull) as not detected, but through our thorough analysis (and manually seeding and executing the mutants), we confirmed that these two mutants were in fact detected, but not reported by the tool. We determined that the remaining three mutants were equivalent.

Next, we used differential testing to determine a baseline for the number of faulty student-written implementations. A convenient property of this assignment is that the methods students were required to implement have a relatively small number of legal inputs (since there are only 52 playing cards in a standard deck). As such, we wrote a small program that conducts exhaustive differential testing against the instructor-written implementation. That is, we compare the return value of each method in the implementation under test to the return value of the instructor's implementation of that same method for every legal combination of input arguments. If the return values do not match, the test reports a failure. Using this strategy, we detected 558 student implementations (out of 768 total implementations) as faulty, which we note is the same result as the set of faulty student implementations detected by all the student-written test suites combined.

However, the instructor test suite (including test cases added to maximize its mutation score) only detected 250 faulty student implementations (about 45% of all faulty implementations). We categorized these faulty implementations by examining their source code, writing mutants of the instructor-written implementation by hand that we suspected would be coupled to those faults, and then writing test cases targeted at those mutants to confirm the coupling. If we could not come up with a mutant of the instructor-written implementation, we instead attempted to fix the student-written implementation with a single change that could be undone with a single application of a mutation operator.

Faults requiring stronger mutation operators (2). We found two unique faults that require a stronger version of the argument omission mutation operator discussed in Just et al [14]. This version of the argument omission operator would perform the omission on all calls to the same function within a statement rather than just one.

Faults requiring new mutation operators (1). We found one fault that would be coupled to a mutant that replaces the use of an overloaded Card comparison operator with a comparator function. This operator would require the user to specify a custom list of functions that might be confused with one another. Note that this is somewhat similar to the "Method Expression" operator supported by Stryker.NET (C#) and Stryker4s (Scala).

Faults requiring an existing mutation operator not supported by Mull (5). We found five unique faults that require a mutation operator that Mull does not support but that another mutation analysis tool does. Four of these would be coupled to a mutant generated with Stryker's "Conditional Expression" operator. One fault would be coupled to a mutant generated with a comparison operator replacement mutation operator that operates at the AST level. The Card class includes overloaded comparison operators, and Mull does not apply mutations to uses of overloaded operators.

Faults requiring a mutant generated from another implementation (7). We found seven unique faults that would be coupled to a mutant generated from an implementation with a different structure from that of the instructor-written implementation. That is, the algorithmic approaches in these implementations were fundamentally different from that of the instructor-written implementation, making these faults impossible to represent with a single mutation of the instructor implementation. Four of these would require Stryker's "Conditional Expression" mutation operator, and the other three

would require the argument omission mutation operator discussed in Just et al [14].

Faults not coupled to mutants (0). In our investigation, we did not find any faults in student-written Game Card implementations that could not be coupled to a mutant using an existing, new, or stronger mutation operator or by generating a mutant from an implementation with a different structure than that of the instructor-written implementation.

4.2.2 Not-Mutant-Coupled Game Player Faults. We found 26 unique faults in student-written Game Player implementations that are not coupled to a mutant generated by Mull from the instructor-written implementation. We note that while eight of these faults have the same root cause, we count them separately because they are detected by different test inputs. These faults were spread across most of the methods in the Player class.

Unlike the instructor-written test suite for Game Card, the unmodified instructor test suite for Game Player had the maximum possible mutation score (we determined that all of the undetected mutants were equivalent). Using the set of faulty student implementations detected by all the student-written test suites combined as a baseline for the total number of faulty student implementations, we find that the instructor test suite for Game Player detects 369/518 faulty student implementations (about 71%).

Faults requiring stronger mutation operators (2). We found two unique faults that require stronger mutation operators. The first of these would involve replacing the right-hand side of an assignment expression with a default-constructed value of the type of the variable being assigned to. While Mull does support replacing primitive values with chosen defaults, it does not support replacing class objects in this way. The second of these would remove an "outer function call" where a single value is passed as an argument to a function that returns a value of that same type.

Faults requiring new mutation operators (0). We did not identify any faults in Game Player implementations that would require a new mutation operator.

Faults requiring an existing mutation operator not supported by Mull (9). We found nine unique faults that require existing mutation operators that Mull does not support. Four of these require the argument omission operator discussed in Just et al. [14], two of these require Stryker's "Conditional Expression" mutation operator, two require Stryker's "Statement Removal" operator, and one would require the "Constant Replacement" operator offered by the PIT mutation analysis tool [22].

Faults requiring a mutant generated from another implementation (1). We identified one fault that would require a mutant generated from an implementation with a different structure than the instructor-written implementation. This implementation uses while loops instead of simple conditionals and then uses `return` statements to break out of the loop in its first iteration when certain conditions are met. This implementation omits an `else` branch that would only need to contain a `return` statement in order to be correct. Without that return statement, the function loops forever in certain cases. Therefore, we could generate a mutant from a corrected version of this implementation using Stryker.NET's statement removal mutation operators.

Faults not coupled to mutants (14). We found 14 unique faults that are not coupled to any mutants. Eight of these are caused the same fundamental flaw in the design of the implementation (omitting the concept of "not yet found" in a linear search), but are detected by different inputs. The remaining six of these faults typically involve incorrect added logic or incorrect assumptions about the invariants of the Player class and its methods and therefore are not possible to represent with a standard mutation operator. In one such fault, the implementation checks whether a card matches either of two suits rather than just a single suit. Since this fault is due to additional code rather than omitted or changed code, it cannot be coupled to a mutant.

The next fault uses a sentinel value to indicate whether certain array indices are empty (rather than maintaining a contiguous array), but the sentinel value is a valid Card object. Although that particular Card value is not used in the top-level game implementation (cards with ranks below nine are not used in Euchre), the contract of the Player class does not prohibit it from being used. Another fault is caused by a value being modified in a case where the Player contract explicitly states that it should not be modified. The value to be modified is paired with a boolean, and the top-level game implementation does not check the value if the boolean is false. The last three of these faults rely on the assumption that the player will have exactly five cards in their hand when certain methods are called. Although this assumption will hold in the context of the top-level game implementation, the contracts of these methods do not state this as a precondition. It is worth noting that none of the instructor-written manually-seeded faults nor the instructor test suite exercise the behavior of these last five faults, which suggests that the instructors did not find it meaningful to evaluate students on these edge cases.

4.2.3 RQ2 Conclusions. For the Game Card and Game Player assignments (recall that these were the only two assignments in which students submitted both implementations and test cases), we found a moderately strong correlation between mutation score and real fault detection rate. After further investigation, it became clear that the strength of correlations was weakened primarily by a small number of unique faults that were not coupled to a mutant and that were not evenly distributed throughout student-written implementations. We note that the set of mutation operators supported by mutation analysis tools is a very important factor in their ability to produce faults that are representative of real student faults. This evidence suggests that mutation score is a reasonably good proxy for faulty student implementation detection in an all-pairs graded scenario even though we only generated mutants from a single instructor-written implementation. We also note that although mutation analysis tools sometimes generate mutants that are not unique, the total number of mutants does not increase relative to the number of students. In all-pairs test suite grading, the number of faulty implementations grows with the number of students, which increases the extent to which unevenly-distributed faults can skew the grading results.

Furthermore, the not-mutant-coupled faults that we discovered are also *not coupled to any manually-seeded faults*. In some cases,

this suggests that the instructors did not find certain faults to be meaningful. In other cases, an implementation with a different structure would have been required to create such manually-seeded faults. The remaining faults could have been manually seeded into the instructor-written implementation but were not, which may suggest that this was an oversight by the instructor.

Complementary to the findings of prior work, our analysis of not-mutant-coupled faults supports the notion that generating mutants from a broader range of instructor-written or (correct) student-written implementations would strengthen these results [29].

5 DISCUSSION

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

5.1 Implications for Researchers

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

There is a growing trend of using mutation analysis 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 resources required to run mutation analysis tools. An experimental design that examines mutants generated from multiple implementations of the same specification 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.

5.2 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 instructor, TA, or student implementations). Using an off-the-shelf mutation analysis 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 [24]. Additionally, instructors can use mutation analysis 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 analysis 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 (e.g., the number of mutants changes with the length of the implementation), it may be worth revisiting the question of whether students should be encouraged to use mutation analysis on their own implementations, outside of the assignment submission feedback loop. Prior work suggests that students benefit from frequent, actionable feedback [8, 18, 29], and teaching students how to apply mutation analysis 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 [23] 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 analysis tools in such a context.

Finally, our findings suggest that mutation analysis tools have untapped potential in educational settings, and we look forward to engaging with the community on this topic. Publicly-available information about the assignments we used in our evaluation can be found with our supplementary materials [19] so that other instructors can use them as a reference for how to structure future assignments that involve evaluating student test suite quality.

5.3 Implications for Tool Builders

Tool builders may be interested in providing better support for educational applications of mutation analysis, since ease of adoption for instructors may improve the visibility of those tools. Mutation analysis 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 analysis to grade student test suites, it is more useful for the mutation analysis tool to support a distinct "mutant generation" phase, where an instructor can determine which mutants should be executed on subsequent executions of student code. Stryker and Mull both support reporting their results as a JSON file that follows the Mutation Analysis 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 [15]). For example, mutation analysis tools could support comparing the mutation scores of multiple test suites so that software developers could examine how a test suite evolves over time.

5.4 Threats to Validity

Construct: Are we asking the right questions? Our research questions are based on established research questions from the mutation analysis 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. 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 4.2 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 and conclusions 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 three 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 analysis tools. While we are not permitted to release student data or submissions, we have made our analysis scripts publicly available so that other researchers may replicate our work using a different set of student submissions [19].

6 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 [12, 16]. It is possible to combine several coverage criteria to better evaluate test suite effectiveness [36]. Jia and Harman present a survey on mutation analysis [13], which Just et al. [14] show is correlated with real-fault detection, even after controlling for coverage. Mutation score is also correlated with defect density [37]. This supports several applications of mutation analysis, including test suite reduction [26–28].

The effectiveness of mutation analysis depends on the kinds of mutants generated, and there are several ways to improve the mutant generation process [6, 15]. While traditional mutation analysis applies only a single mutation at a time [13], one line of research examines the efficacy of *higher order mutations*, which are generated by applying multiple mutation operations simultaneously [11, 39]. Our methodology of studying mutation analysis 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 analysis tools could also make the correlations that we find even stronger.

There is also evidence that mutation analysis helps programmers write better test suites [21]. Our 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 analysis in educational settings, and it would be interesting future work to study whether exposing students directly to mutation analysis results would result in better test suites.

Clegg et. al examine the extent to which real faults in student-written code are coupled to mutants [4]. They use a "coupling ratio" metric that divides the number of tests that couple a given implementation to at least one mutant by the number of failing test cases for that implementations, and the test suites in their analysis are either instructor-written, hand-written by the authors, or generated using EvoSuite. They conclude that "students' faulty solutions are often coupled to mutants" and therefore that instructors should conduct mutation analysis on the test suites they use to evaluate student source implementations. We believe our study is complementary to this work and differs in several ways. First, our work evaluates the effectiveness of mutants for the purpose of evaluating

students' test suites. Whereas Clegg et al. study only student implementations, we study both student implementations and student test suites. This allows us to draw implications for grading test suites, not only for grading student implementations. We consider two grading scenarios: one where student test suites are graded on their detection of manually-seeded faults (RQ1), and one where student test suites are graded on their detection of faulty student implementations (RQ2). Second, our sample size is significantly larger (2,711 vs. 197) and draws from several teaching institutions and several programming languages. Third, we identify faults in student implementations that are not coupled to a mutant and investigate whether such faults could be replicated with new or strengthened mutation operators.

One potential concern when attempting to generalize these prior studies of mutation analysis 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, 38].

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 a 28% reduction in defects per thousand lines of code after students were given automated coverage feedback [8]. Later work shows that coverage is a poor indicator of student test quality, and instead develops an approach to grading based on mutation analysis [2, 24]. Moreover, in an "all-pairs testing" approach, no significant correlation arises between the fault-detection rate of a test suite and its code coverage or its mutation score [9, 25].

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 seeded manually 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 analysis [29]. Wrenn et al. [40] discuss several flaws with automated assessment of student code and recommend evaluating student tests with multiple implementations. Our work shows that mutation analysis is a scalable way of generating multiple (faulty) implementations and is as effective as having multiple, manually-seeded faulty implementations.

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 analysis 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 analysis should consider evaluating mutants generated from multiple implementations of the same system under test when feasible.

ACKNOWLEDGMENTS

We thank our co-instructors, TAs and students for their cooperation and feedback. This work was funded in part by NSF CCF-2100037, NSF CNS-2100015, and NSF CCF-502916.

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] Benjamin Simon Clegg, Phil McMinn, and Gordon Fraser. 2021. An Empirical Study to Determine If Mutants Can Effectively Simulate Students' Programming Mistakes to Increase Tutors' Confidence in Autograding. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education*. Association for Computing Machinery, New York, NY, USA, 1055–1061. https://doi.org/10.1145/3408877.3432411

[5] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. 2016. PIT: A Practical Mutation Testing Tool for Java (Demo). In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. Association for Computing Machinery, New York, NY, USA, 449–452. https://doi.org/10.1145/2931037.2948707

[6] 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.

[7] 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

[8] 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

[9] 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

[10] 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

[11] 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

[12] Laura Inozemtseva and Reid Holmes. 2014. Coverage is Not Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. Association for Computing Machinery, New York, NY, USA, 435–445. https://doi.org/10.1145/2568225.2568271

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

[14] 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

[15] 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

[16] 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

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

[18] James Perretta and Andrew DeOrio. 2018. Teaching Software Testing with Automated Feedback. In *2018 ASEE Annual Conference & Exposition*. ASEE Conferences, Salt Lake City, Utah. https://peer.asee.org/31062.

[19] James Perretta, Andrew DeOrio, Arjun Guha, and Jonathan Bell. 2022. Supplementary Materials. https://doi.org/10.5281/zenodo.6564504

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

[21] 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)*.

[22] PIT Overview 2022. PIT Overview. https://pitest.org/quickstart/mutators/

[23] 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

[24] 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

[25] 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

[26] 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

[27] August Shi, Alex Gyori, Suleman Mahmood, Peiyuan Zhao, and Darko Marinov. 2018. Evaluating Test-Suite Reduction in Real Software Evolution. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 84–94. https://doi.org/10.1145/3213846.3213875

[28] August Shi, Tifany Yung, Alex Gyori, and Darko Marinov. 2015. Comparing and Combining Test-Suite Reduction and Regression Test Selection. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 237–247. https://doi.org/10.1145/2786805.2786878

[29] 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

[30] COMPSCI 220 Course Staff. 2019. COMPSCI 220, Programming Methodology. https://umass-compsci220.github.io/2019F/

[31] CS 4530 Course Staff. 2022. CS 4530, Fundamentals of Software Engineering. https://pages.github.ccs.neu.edu/CS5500-CourseMaterials/CS4530-CS5500-Fall2020/index.html

[32] EECS 280 Course Staff. 2022. EECS 280, Programming and Intro Data Structures. https://eecs280staff.github.io/eecs280.org/

[33] Eike Stein, Steffen Herbold, Fabian Trautsch, and Jens Grabowski. 2021. A new perspective on the competent programmer hypothesis through the reproduction of bugs with repeated mutations. https://arxiv.org/abs/2104.02517

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

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

[36] 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)*.

[37] 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

[38] 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

[39] 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

[40] 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