Who Judges the Judge: An Empirical Study on Online Judge Tests

Kaibo Liu
liukb@pku.edu.cn
Peking University
Beijing, China

Yudong Han
hanyd@pku.edu.cn
Peking University
Beijing, China

Jie M. Zhang
jie.zhang@kcl.ac.uk
King's College London
London, United Kingdom

Zhenpeng Chen*
zp.chen@ucl.ac.uk
University College London
London, United Kingdom

Federica Sarro
f.sarro@ucl.ac.uk
University College London
London, United Kingdom

Mark Harman
mark.harman@ucl.ac.uk
University College London
London, United Kingdom

Gang Huang
hg@pku.edu.cn
Peking University
Beijing, China

Yun Ma*
mayun@pku.edu.cn
Peking University
Beijing, China

ABSTRACT

Online Judge platforms play a pivotal role in education, competitive programming, recruitment, career training, and large language model training. They rely on predefined test suites to judge the correctness of submitted solutions. It is therefore important that the solution judgement is reliable and free from potentially misleading false positives (i.e., incorrect solutions that are judged as correct). In this paper, we conduct an empirical study of 939 coding problems with 541,552 solutions, all of which are judged to be correct according to the test suites used by the platform, finding that 43.1% of the problems include false positive solutions (3,700 bugs are revealed in total). We also find that test suites are, nevertheless, of high quality according to widely-studied test effectiveness measurements: 88.2% of false positives have perfect (100%) line coverage, 78.9% have perfect branch coverage, and 32.5% have a perfect mutation score. Our findings indicate that more work is required to weed out false positive solutions and to further improve test suite effectiveness. We have released the detected false positive solutions and the generated test inputs to facilitate future research.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Online judge platform, software testing, test assessment

*Corresponding authors

ISSTA 2023, 17-21 July, 2023, Seattle, USA
© 2023 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/XXXXXXX.XXXXXXX

ACM Reference Format:
Kaibo Liu, Yudong Han, Jie M. Zhang, Zhenpeng Chen, Federica Sarro, Mark Harman, Gang Huang, and Yun Ma. 2023. Who Judges the Judge: An Empirical Study on Online Judge Tests . In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2023).* ACM, New York, NY, USA, 13 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Online Judge (OJ) systems are designed for providing coding tasks and then evaluating the solutions submitted by users. The tasks are typically challenging coding problems (e.g., those with tricky corner cases and complex engineering trade-offs) that include a narrative statement of requirements, together with example inputs and outputs. OJ platforms have been widely adopted to support education [21, 34, 67], code contests [57, 66], recruitment [66], and programming training [18]. OJ problems and solutions are also essential resources for the training data of large language models, such as ChatGPT [49], AlphaCode [36] and Github's Copilot [1].

Take recruitment as an example, big companies such as Amazon, Linkedin, and Bytedance use OJ platforms to source, screen, and interview developers. In addition, many OJ platforms help would-be software engineers train against a set of typical coding interview problems [3, 4, 27, 44, 70]. OJ platforms thereby form a *critical gatekeeper* that partly determines the composition of the Software Engineering profession: they contribute to the processes by which software engineers are educated, trained, and assessed, and the criteria for joining the Software Engineering profession. It is therefore essential to understand the properties of the OJ process from multiple perspectives.

Existing research on OJ platforms primarily focuses on applications in Software Engineering education [8, 21, 34, 57, 67, 79]. However, there have been very few studies on the issues that might impact the efficacy of the online judgement process. It is here that Software Testing research has a role to play in finding ways to identify and improve candidate solutions, and the test suites used to evaluate them. In particular, OJ platforms use predefined test suites to automatically judge the correctness of submitted solutions [66].

Software testing thereby has a direct, profound, and lasting impact on the correct operation of OJ systems, and on the advice and guidance they provide.

There is no previous work that seeks to systematically understand and assess OJ test effectiveness. Forišek et al. [20] have previously highlighted the problem that software testing might prove to be unreliable for code contests such as the ACM International Collegiate Programming Contest (ICPC) and the International Olympiad in Informatics (IOI) competitions. However, although they identified these important potential limitations, there has, hitherto, been no further empirical study. Based on the current literature, we thus have no results on OJ test suite effectiveness. Nor do we have any assessment of the prevalence of false positive solutions (i.e., incorrect solutions that are judged as correct) among those offered to train, recruit and guide the Software Engineering practitioner community.

With the rise of using large language models for code generation, it is more and more vital to guarantee that the OJ solutions in the training data are correct. In fact, we have already observed a case where ChatGPT provides an incorrect solution which is exactly identical to one of our detected false positive solutions[1]. This case further reveals the importance of understanding and detecting false positive solutions in OJ platforms.

This paper tackles these important gaps in the current literature on OJ platforms. Based on a large-scale empirical study of 541,552 code solutions to 939 publicly available coding problems, the paper assesses OJ test suite effectiveness with respect to 4,703,239 mutants. All solutions have passed the predefined tests and are thus judged, by the OJ platform, to be correct. We collect our dataset from AtCoder [27], a well-known OJ platform that makes test suites and model solutions publicly available. We measure the line coverage, branch coverage, and mutation score for all the solutions we collect, and calculate their correlation with program features such as lines of code and the number of tests. We then use test generation and differential testing to detect whether there remain false positive solutions among those offered. Finally, we report on the degree to which widely-studied coverage techniques are effective at exposing any weaknesses in the OJ test suites.

Our study reveals the following key findings: **1)** Overall, OJ tests have very good reliability according to coverage and mutation score: 91.5% of the OJ solutions have full line coverage, and 85.8% have full branch coverage. Although only 42.8% of the solutions have full mutation scores, the majority (e.g., 95.3% for Python) of the survived mutants are equivalent mutants. **2)** The correlation between line coverage, branch coverage, and mutation score highly depends on the programming language that a solution adopts. Additionally, the number of tests is observed to have a negative correlation with coverage and mutation score on the OJ solutions that we study. **3)** Despite the high coverage and mutation score (modulo equivalent mutants), we detect 3,700 bugs, which are ignored by the existing OJ tests. A large proportion (43.1%) of the OJ problems that we study are found to contain false positive solutions. **4)** A further investigation of the detected false positive solutions indicates that 88.7% of the false positive solutions have full line coverage, 79.0% have perfect branch coverage, and 37.8% have full mutation scores.

[1]The details for the false positive solution can be found on our homepage [37].

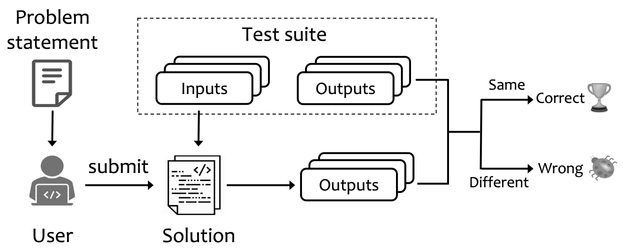


Figure 1: Online Judge Platform Workflow.

These observations reveal the limitations and challenges of the existing test assessment techniques as well as a series of opportunities to improve them.

To conclude, this paper makes the following contributions:

- An empirical study on the coverage and mutation score for online judge solutions that have passed the predefined tests.
- An analysis of uncovered code and survived mutants.
- An investigation on the effectiveness of coverage and mutation score in avoiding false positive solutions.
- An open-source benchmark with false positive solutions to facilitate software bug-related research [37].

The rest of the paper is organized as follows. Section 2 introduces the preliminaries of our study. Section 3 describes our research questions and experimental setup. Section 4 answers the research questions based on the experimental results we obtained. Section 5 discusses the threats to the validity of this study and the implications of our results. Section 6 summarizes related work, followed by concluding remarks in Section 7.

2 PRELIMINARIES

2.1 Online Judge Platforms

OJ platforms provide coding problems (namely *OJ problems*) for users, and then automatically judge the correctness of the solutions submitted by the users. Figure 1 presents the workflow of a generic OJ platform.

An OJ problem mainly consists of a *problem statement* and a predefined *test suite*. The *problem statement* introduces the coding tasks and specifies the input and output formats (e.g., a series of integers) of the problem. In addition, it provides the *input constraints* (e.g., the input should be a positive integer if it represents the number of objects). The problem statement often provides example inputs and outputs to guide users. The users can specify the programming language they want to use, write the code to solve the OJ problem, and then submit their own solutions to the OJ platform. Once a solution is submitted, the OJ platform compiles the solutions (if necessary) according to the specified language and judges the solution's correctness by using a predefined test suite. The predefined test suite is not visible to users. A submitted solution is considered correct only if it can produce correct outputs for all the test cases in the test suite [45].

The test suites are therefore the OJ platform "judges", who determine whether a submitted solution is correct. A vital yet underexplored problem is: How reliable are these "judges"? Is it likely that the test suites misjudge the correctness of some solutions? This paper aims to answer such questions. If the test suite judges a buggy solution as correct, we call the solution a *false positive solution*.

2.2 Test Assessment Metrics

The main goal of this study is to assess the reliability of test suites in OJ platforms. To this end, we use three test assessment metrics that are most widely studied in the literature: line coverage (LC), branch coverage (BC), and mutation score (MS) [32, 69].

- **Line coverage** measures the percentage of executed lines of the source code against the total lines of code when running the test suite. Its assessment of the test suite is based on the truth that bugs in the uncovered code can never be detected by the test suite.
- **Branch coverage** is similar to line coverage. It measures the percentage of covered branches. For example, an `if` statement in the source code creates two execution branches, and a reliable test suite is expected to cover both branches.
- **Mutation score** is the score used in mutation testing [50]. Mutation testing makes minor changes to the source code by using different mutation operators. The modified programs are called *mutants*. A mutant is called an "equivalent mutant" if its semantic is equivalent to the original program, despite being syntactically different. If the test suite is not able to differentiate a mutant and its original program, we call this mutant a "survived" mutant. Otherwise, we call it a "killed" mutant. The mutation score is the percentage of killed mutants against the total number of non-equivalent mutants. In practice, due to the challenges in automatic equivalent mutant detection, mutation score is often calculated as the percentage of killed mutants against the whole set of mutants in the existing mutation testing tools [9, 13, 46]. Considering that uncovered mutants cannot be killed, we use two types of mutation scores in our study, i.e., the original mutation score and the covered mutation score. The former is computed based on all mutants (i.e., mutants applied to any line of the program); the latter is computed based on only those mutants applied to covered lines of the program (i.e., lines that are executed by the test suite).

3 METHODOLOGY

This section introduces our research questions, data collection process, test assessment techniques, and false positive identification approach that we use to detect bugs that have been overlooked by existing OJ test suites.

3.1 Research Questions

We aim to answer the following research questions in our study.
RQ1: What coverage and mutation score are achieved by the OJ test suites used to judge coding solutions?
RQ1.1: What coverage is achieved by the OJ test suites and what is the cause for uncovered code?
RQ1.2: What mutation score is achieved by the OJ test suites and what is the cause for survived mutants?
RQ1.3: How do coverage and mutation score correlate with other problem characteristics (e.g., problem difficulty, the number of tests, and lines of code)?

Table 1: Sizes of solutions and tests in the dataset we collect.

Measurement	Min	Max	Average
Lines of codes (Python)	1	413	13.5
Lines of codes (Java)	1	4,507	68.0
Lines of codes (C++)	1	1,066	31.2
Sizes of tests	2	148	30.9

RQ2: Are there false positive solutions that are judged as correct by OJ test suites but are actually buggy?
RQ3: How do existing test assessment techniques contribute to the identification of false positive solutions?

3.2 Dataset Collection

To answer our research questions, we need to analyze OJ problems as well as user-submitted solutions and predefined OJ test suites. To this end, we use data from AtCoder [27] for the following reasons: 1) it is a well-known OJ platform with more than 150,000 active users; 2) its data has been widely used for research purposes [23, 36, 40], and is included in the well-known CodeNet dataset [55] and CodeContests dataset [36]; 3) it is the **only** popular OJ platform that makes its full set of predefined tests available to download[2].

We downloaded the AtCoder problems and the corresponding accepted solutions (i.e., solutions that are judged as correct by the predefined test suites) from CodeContest, where duplicate problems and solutions were already removed [36, 55]. This data consists of 939 coding problems and 541,552 accepted solutions, including 168,909 Python solutions, 138,401 Java solutions, and 234,242 C++ solutions. In our study, we focus on solutions written in Python, Java, and C++, because these are the three most popular languages in OJ platforms [36], accounting for 96.7% of all solutions available.

We then collect the predefined test suites for each problem from the official website provided by Atcoder [28]. Among the 939 coding problems downloaded from CodeContest, AtCoder website does not provide full test suites for 88 problems that are outdated. We thus use the remaining 851 problems in RQ1 and RQ3 for coverage and mutation score analysis, whereas we use the whole set of problems in RQ2 where the availability of tests does not affect the results. Table 1 summarizes the sizes (min/max/avg) of solutions and test suites in our dataset.

For each solution, we also collect the difficulty of the problem, the number of predefined tests, and the lines of code. We obtain the difficulty for each coding problem calculated by a third-party website [47], which is estimated statistically according to the level of users who solve the problem successfully during the contest.

3.3 Code Coverage Analysis

To answer RQ1, we first measure the line coverage and branch coverage of the test suites. Line coverage and branch coverage are the two most commonly-used coverage metrics in academia and industry [42].

[2]CodeContests provides test cases for other OJ platforms, but the test cases are only example tests in the problem statements and a subset of hidden test cases that are made available at the evaluation result pages once a contest is finished, which are different from the predefined full set of tests adopted by the platforms.

For Python solutions, we use Coverage.py [48], a widely-used tool for measuring the coverage of Python code. For C++ solutions, we use the source-based code coverage feature of clang [35] in order to measure line coverage and branch coverage. For Java solutions, we use JaCoCo [30], which is well-known for its integration with the Eclipse workbench.

We also manually analyze the uncovered code for each language. We randomly sample the solutions with uncovered code to ensure a 95% confidence level and a 5% confidence interval following previous work to save manual efforts while also obtaining statistically significant conclusions [12, 38, 75]. Then, the first two authors, who have five and eight years experience of participating in OJ contests, manually analyze the sampled solutions. The analysis consists of two rounds. In the first round, the two authors independently analyze each solution and identify the cause for the uncovered code. In the second round, they discuss their results jointly and resolve conflicts by introducing other authors as arbitrators.

3.4 Mutation Score Analysis

To answer RQ1, we also need to measure the mutation score achieved by the OJ test suites. We use mutmut [9], a tool having over 600 stars on GitHub, to conduct mutation testing for Python solutions. Mutmut is widely adopted for mutation testing research on Python [15, 22, 24, 71]. We use Mull [17, 46] to conduct mutation testing for C++ solutions, which is also a widely used tool with more than 600 stars on GitHub. We set the mutation operator as "cxx_default", the most common setting for Mull [17, 31]. For Java, we use PITest, a state-of-the-art mutation testing tool with 1.4k stars on GitHub [13, 65]. As suggested by PITest, we configure the mutation operators as "DEFAULTS", which is stable and tends to generate fewer equivalent mutants [53].

For survived mutants that are not killed by the test suites, we check whether they are equivalent mutants by manual analysis. The mutation tools we use for Java and C++ solutions conduct byte-code level and intermediate representation level mutations respectively and do not produce physical mutants. We thus focus on the survived Python mutants that can be obtained directly through the Python mutation tool.

Similar to what we explained in Section 3.3, we randomly select 381 mutants from 74,447 survived Python mutants to ensure a 95% confidence level and a 5% confidence interval. Then the first two authors manually check the survived mutants following the same procedure of analyzing uncovered code (Section 3.3).

3.5 False Positive Solution Identification

To identify false positive solutions, we randomly generate extra test inputs and then use differential testing to detect the solutions that yield different outputs from the majority of the solutions. AlphaCode [36] also uses this method to identify the false positive solutions it generates for OJ problems.

3.5.1 Test Input Generation. To answer RQ2, we randomly generate 100 inputs for each OJ problem. The input types of OJ problems are often number, vector, and graph under specified constraints. To generate a random number, we sample the number uniformly from a range specified in the constraints. To generate a random vector, such as a string or an array of integers, we sample each element in this vector uniformly from the range specified in the constraints. To generate a random graph, we use Cyaron [39], which builds a random graph by repeatedly adding an edge between two random vertices. For example, if the input constraints of the A+B problem (See Figure 2) are $0 \leq A \leq 10^5$ and $0 \leq B \leq 10^9$, our random input generator samples an integer uniformly from 0 to 10^5 as A, and samples an integer uniformly from 0 to 10^9 as B.

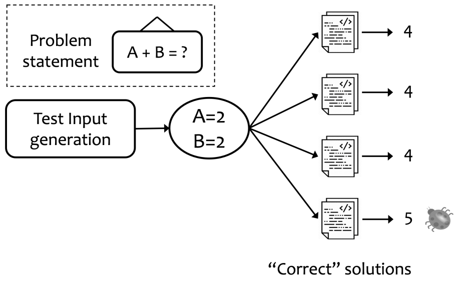


Figure 2: The procedure of detecting false positive solutions.

3.5.2 Test Oracle Generation. For the newly generated inputs, it is difficult to automatically identify the explicit oracles. To tackle the oracle problem, following AlphaCode [36], we use *differential testing* [41]. Differential testing runs the same set of inputs to a series of programs with the same functionality and detects the difference in their outputs.

Specifically, for each OJ problem, we feed the generated inputs to all solutions that are judged as correct by the OJ test suites, and then collect the outputs of these solutions. The solutions whose outputs are different from the majority solutions are considered false positive solutions.

Figure 2 shows the procedure of differential testing with an example of the A+B problem. The inputs are two integers A and B, and the user is asked to compute the result of $A + B$. We feed a generated input "$A = 2, B = 2$" to the user-submitted solutions that are judged as correct. As a result, the last solution outputs "5" while other solutions output "4". Since this OJ problem has only one correct answer, we consider the last solution a false positive solution. There are multi-answer OJ problems, where there are multiple correct answers. To this end, we manually filter out 48 (5.1%) multi-answer problems for RQ2 and identify false positive solutions for the remaining 891 OJ problems. The validity of this approach is further investigated in Section 4.2.

4 RESULTS

This section provides results and analysis to answer our RQs.

4.1 RQ1: Code Coverage and Mutation Score

RQ1 explores the coverage and mutation score achieved by the predefined OJ test suites. We also explore the correlation between coverage, mutation score and other program characteristics (i.e., problem difficulty, lines of code, and the number of tests) to understand potential factors that may influence the assessment results.

4.1.1 RQ1.1 - Code Coverage.

Solution-level coverage analysis: We first measure line coverage

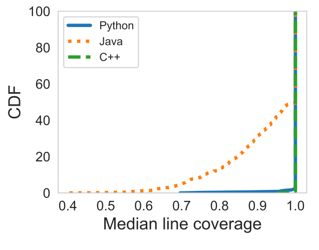
Table 2: RQ1.1: Ratio of solutions with full line coverage and branch coverage. Overall, 91.5% of the solutions are fully line covered, and 85.8% are fully branch covered.

Language	Line coverage	Branch coverage
Python	98.4%	96.3%
Java	78.2%	74.2%
C++	97.2%	88.3%
Total	91.5%	85.8%

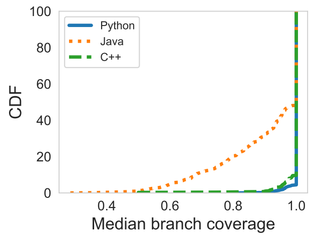
and branch coverage at the solution level. Table 2 shows the ratio of solutions with full line coverage and branch coverage for each language. We observe that the majority of the Python and C++ solutions are fully covered: 98.4% of Python solutions and 97.2% of C++ solutions are fully line covered; 96.3% of Python solutions and 88.3% of C++ solutions are fully branch covered; in contrast, the coverage for Java solutions are relatively low: 78.2% of Java solutions are fully line covered, and 74.2% of Java solutions are fully branch covered. This may be due to the fact that Java is more verbose [73] and as such it might be more difficult to fully cover Java code with respect to fully covering Python and C++ code when using the same test suite.

Problem-level coverage analysis: We also study the code coverage at the problem level. There are hundreds of solutions for each problem, and we explore the standard deviation of line/branch coverage for these solutions. Fig 3 shows the Cumulative Distribution Function (CDF) results. It is interesting to observe that the coverage results for different solutions targeting the same problem are very similar for Python and C++: The majority of the problems' coverage standard deviation is below 0.1. Nevertheless, for Java, over 40% of the problems have a standard deviation larger than 0.2. In the following, we dig deep into the uncovered code and investigate the cause and categories of uncovered code.

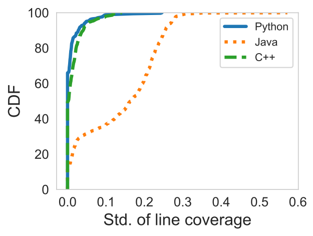
Analysis of uncovered code: Our coverage analysis reveals 2,145 Python solutions, 23,366 Java solutions and 8,094 C++ solutions with an imperfect line or branch coverage. Following the procedure introduced in Section 3.3, we randomly select 257 Python solutions, 372 Java solutions, and 302 C++ solutions to ensure a 95% confidence level and a 5% confidence interval. We then manually analyze these sampled solutions, which led us to discover three main types of uncovered code: *1) Missing Branch* means the uncovered part is a branch that affects program behaviours but is not executed during testing. Errors in this type of uncovered code will not be exposed by the existing tests, thereby yielding a false positive solution. *2) Template Code* means the uncovered code is from user templates. Some users copy chunks of templated code into all of their solutions to improve efficiency during coding competitions. The code templates are pre-written codes that can be widely used for different problems, such as common I/O parsers, common mathematical functions, and popular data structure classes. The unused pre-written code for a particular problem then becomes dead code that cannot be covered. *3) Debugging Code* is the code users wrote for debugging before submission. The users may forget to delete this code after debugging, or simply decide to keep it because it does not affect the behaviour of the functional code.



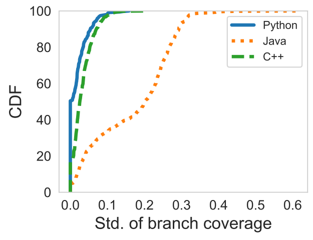
(a)



(b)



(c)



(d)

Figure 3: RQ1.1: CDF of problem-level median coverage and standard deviation. (a) CDF of median line coverage of problems (b) CDF of median branch coverage of problems (c) standard deviation of median line coverage of problems (d) standard deviation of median branch coverage of problems

```
//Example of Missing Branch:
    if (k % 2 == 0) {
            long ans = a-b;
            if (ans > th || -ans > th) {
                System.out.println("Unfair");
            } else {
                System.out.println(ans);
            }
    }
//Example of Template Code:
    long gcd(long a, long b){
    return b == 0 ? a : gcd(b, a % b);
    }
    int lcm(int a, int b){
    return a / gcd(a, b) * b;
    }
//Example of Debugging Code:
    if (tt0 < 0 || tt1 < 0 || tt0 + tt1 != n) {
        throw new RuntimeException();
    }
```

Listing 1: Three main types of uncovered code in Java.

Listing 1 shows examples of the three main types of uncovered code in the Java language, and Table 3 shows the ratio of solutions that contain each type of uncovered code. A solution may have more than one type of uncovered code, and thus the ratios in a row

Table 3: RQ1.1: Category of uncovered code. Template code is the main reason of uncovered code.

Language	Missing Branch	Template Code	Debugging Code	Others
Python	42.8%	56.3%	4.7%	0.8%
Java	8.2%	93.2%	0.5%	1.1%
C++	31.8%	68.0%	0.99%	2.0%

Table 4: RQ1.2: Ratio of solutions with full mutation scores. Overall, the test suites achieve full original mutation scores for 43.9% of solutions and achieve full covered mutation scores for 42.8% of solutions.

Language	Original mutation score	Covered mutation score
Python	58.2%	58.4%
Java	40.1%	43.6%
C++	35.6%	35.7%
Total	42.8%	43.9%

may not necessarily sum up to 1. Interestingly, we observe that the majority of the uncovered code in Java is caused by Template Code. One possible reason is that Java is a verbose language [73], and users have a relatively stronger motivation to copy pre-written code to save coding time compared to other languages. We also observe that over half of the uncovered cases in Python and C++ are caused by Template Code or Debugging Code, which are not supposed to change a program's behaviour and are unlikely to yield false positive solutions. In other words, test suites are not expected to cover them. These observations pose concerns regarding the effectiveness of coverage criteria in test suite assessment, which are further explored in RQ2 (Section 4.3).

4.1.2 RQ1.2 - Mutation Score. We conduct mutation testing for all the collected OJ solutions. For each solution, we collect two types of mutation scores: original mutation score and covered mutation score. The former is calculated as the ratio of killed mutants against the total number of mutants; the latter is the ratio of killed mutants against only those mutants on covered lines of the program.

Solution-level mutation score analysis: Figure 4 illustrates the results for mutation score distribution. The last column of Table 4 also shows the ratio of solutions with full mutation scores. For the original mutation score (Figure 4.(a) and (c)), Java solutions have lower mutation scores than Python and C++. This is as expected, because Java solutions have the lowest coverage as shown in Section 4.1.1, yet mutants on uncovered code cannot be killed. Once we count only the mutants on the covered code (Figure 4.(b) and (d)), the mutation score of Java solutions increases remarkably.

When comparing mutation score with line coverage and branch coverage, we observe that the values of mutation score are much lower than the two coverage criteria. In particular, we can observe from Table 4 that only around 60% of the Python solutions, 40% of the Java solutions, and 35% of the C++ solutions have a full mutation score. These observations indicate that achieving full mutation coverage is more difficult than achieving full statement or branch coverage.

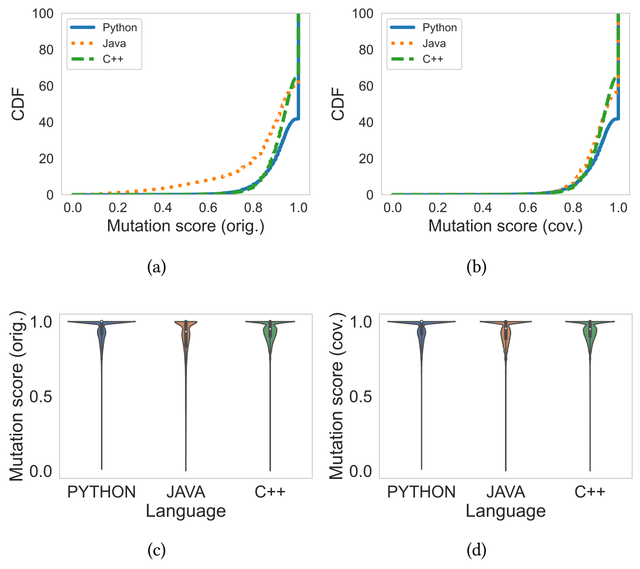


Figure 4: RQ1.2: CDF and violin plot of solution-level mutation scores (original and covered). (a) CDF of mutation score (original) of solutions. (b) CDF of covered mutation score (covered) of solutions. (c) Violin plot of mutation score (original) of solutions. (d) Violin plot of mutation score (covered) of solutions.

Problem-level mutation score analysis: Figure 5 shows the median mutation score as well as the standard deviation for different problems. Compared to Figure 3, the mutation score of different solutions for one problem is more diverse than the coverage. For example, the ratio of problems whose standard deviation being below 0.1 is 85% for mutation score, but almost 100% for line coverage.

Analysis of equivalent mutants: In mutation testing, a mutant can be equivalent, which means the mutation does not bring any semantic changes [32, 50]. The existence of equivalent mutants brings noise to the use of mutation score in assessing test suites because equivalent mutants cannot be killed. In this section, we manually check the survived mutants in order to analyze the prevalence and categories of equivalent mutants. We only focus on the mutants that are covered, because they provide extra information that code coverage cannot provide.

As we describe in Section 3.4, we manually check 381 Python survived mutants. To better understand these mutants, we classify them into three types: *inequivalent mutants*, *universally-equivalent mutants*, and *constraint-equivalent mutants*. If a mutant's behaviour is semantically equivalent to the original program, we classify it as *universally-equivalent* (which is the same as the definition of equivalent mutants in traditional mutation testing). If a mutant is semantically different from the original program, but the difference exists only for inputs that disobey the input constraints, we classify it as *constraint-equivalent*. In other words, constraint-equivalent mutants are equivalent to the original program under all the inputs satisfying the constraints. This type of equivalent mutant is important because different from traditional programs, an OJ program is

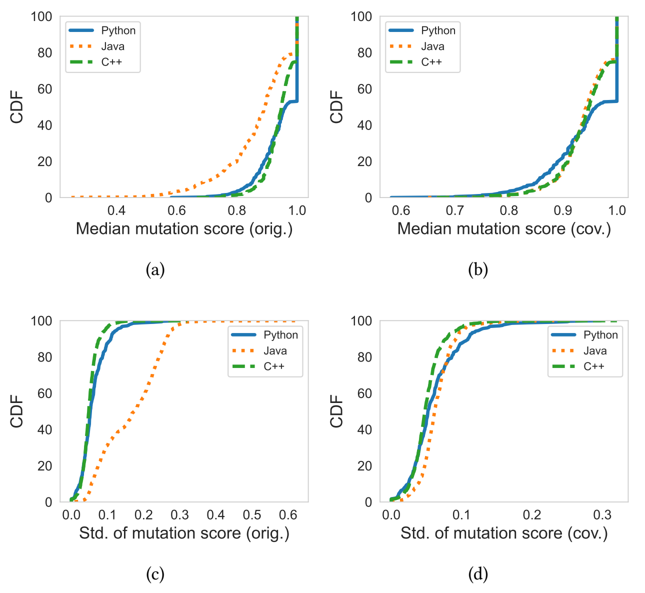


Figure 5: RQ1.2: CDF of problem-level median mutation score (original and covered) and standard deviation (stdv). (a) CDF of median mutation scores (original) of problems (b) CDF of median mutation scores (covered) of problems (c) stdv of median mutation scores (original) of problems (d) stdv of median mutation scores (covered) of problems.

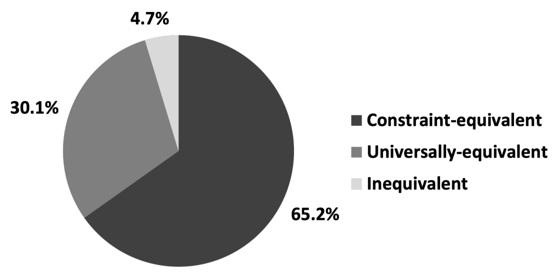


Figure 6: RQ1.2: Composition of survived mutants. Overall, 95.3% of survived mutants are equivalent.

coded to behave correctly for only the inputs under the constraints of the problem description (see Section 2.1). Last, if a mutant is semantically different from the original program under any input satisfying the constraints, we classify it as *inequivalent*.

Figure 6 shows the distribution of the three types of surviving mutants. Surprisingly, the majority (95%) of the survived mutants are either universally-equivalent (30%) or constraint-equivalent (65%). Inequivalent mutants, which survived due to inadequate test suites, account for less than 5%.

We suspect that constraint-equivalent mutants are prevalent because the constraints are usually tight, and the inputs that expose the differences between mutants and the original program can easily fall outside the range. For example, there is a problem whose input constraint is a positive or negative integer. The output is the sign of the input integer. A passed solution for this problem is $if(x<0)\{printf(``-");\}else\{printf(``+");\}$, and a survived mutant is to replace $x < 0$ with $x < 1$. If we mutate 0 to 1 in the condition statement, the mutant program is constraint-equivalent, because the test input (i.e., $x = 0$) that causes different behaviours does not satisfy the input constraints.

The large ratio of equivalent mutants will significantly affect the effectiveness of mutation scores in accessing OJ tests. If we use the ratio of equivalent mutants to estimate the mutation score modulo equivalent mutants, the average covered mutation score will be changed from 43.9% (shown in Table 4) to 97.4%. In RQ3 (Section 4.3), we dig deep into the effectiveness of mutation score in test assessment. We also discuss the challenges and research opportunities for mutation testing in Section 5.

4.1.3 *RQ1.3 - Correlation between Different Metrics.*

The correlation between different test assessment metrics has been widely studied by previous work [5, 29, 74]. In this part, we explore the correlation between different metrics on OJ solutions. Our variables are comparable rank variables so we choose Spearman's rank correlation which is suitable for our experiment.

We conduct Spearman correlation analysis for each of the features we introduced in Section 3, including:

- LC: Line coverage of the solution.
- BC: Branch coverage of the solution.
- MS-O: Mutation score with all the mutants.
- MS-C: Mutation score with covered mutants.
- L: Lines of code.
- D: Difficulty of the problem the solution attempt to answer.
- #T: The number of test cases per problem.

Figure 7 shows the correlation results. The p-values of all correlations in the figure are smaller than 1e−5. Following conventional interpretation [59], we regard the correlation coefficient (absolute value) between 0.9 and 1 as a very strong correlation, 0.7–0.89 as a strong correlation, 0.4-0.69 as moderate, 0.1-0.39 as weak, and smaller than 0.1 as negligible. Based on the results obtained, we can make the following primary observations: **1)** line coverage, branch coverage, and mutations core are strongly, or very strongly, correlated with each other for Java solutions, but are weakly correlated with each other for Python and C++ solutions. Most existing work on the correlation between these metrics was conducted on Java programs [29, 74, 76]. This observation indicates that programming language is an important factor that should not be overlooked for such types of correlation analysis. **2)** the number of tests has a negative correlation with code coverage and mutation score, respectively. This is because solutions with more lines of code are more difficult to have high coverage and mutation score, although they have more tests than short solutions. This observation is completely opposite to previous findings where test suite size is either found to have a very strong correlation with coverage [76] or play an important role in the observed correlation [29].

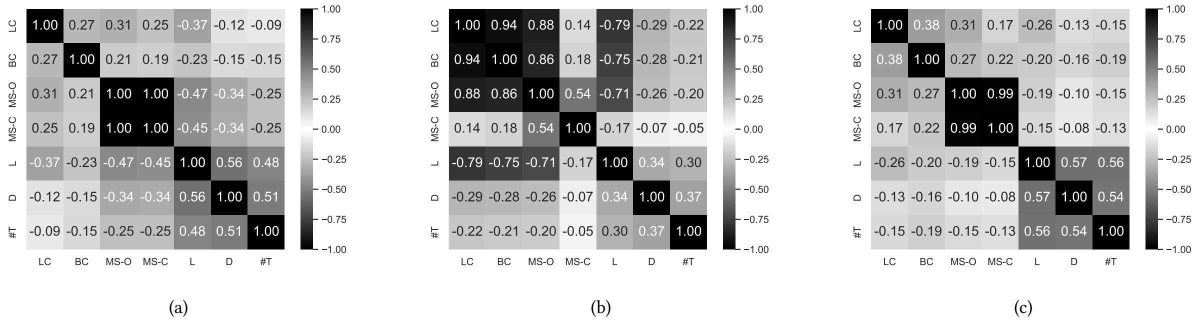


Figure 7: RQ1.3: Spearman's rank correlation coefficient for different variables. (a) Python (b) Java (c) C++. We find the more difficult the problem, the harder for test suite to achieve high coverage and mutation score, even with more test cases.

Answer to RQ1: On average, 91.5% of the OJ solutions have full line coverage, 85.8% have full branch coverage, and 42.8% have full mutation score. However, the majority (95.3%) of survived mutants are equivalent mutants. The correlation between line coverage, branch coverage, and mutation score highly depends on the programming language a solution adopts. Additionally, the number of tests is observed to have a negative correlation with coverage and mutation score on OJ solutions.

4.2 RQ2: Prevalence of False Positive Solutions

From RQ1, we have observed that both coverage and mutation score (modulo equivalent mutants) assess that the existing OJ tests are very strong. In RQ2, we investigate whether their assessment is reliable. To answer this research question, we check whether there are false positive solutions that contain bugs yet they are accepted as correct solutions by the predefined OJ test suites for the 891 OJ problems we investigated herein. The experimental setup details are described in Section 3.5.

Adequacy of the detected false positive solutions: When detecting false positive solutions, we randomly generated 100 inputs within the constraints specified for each problem. Before reaching any conclusions, we explore how the number of generated inputs influences the number of problems that contain false positive solutions as well as the number of false positive solutions. This will give us hints on the reliability of using 100 inputs for false positive solution detection. Figure 8 shows the results. The x-axis is the number of generated inputs. The orange line is the number of problems that are detected to contain false positive solutions, which reaches a plateau when there are around 15 inputs. The blue line is the number of detected false positive solutions, which reaches a plateau when there are around 95 inputs. These observations give us confidence that using 100 inputs per OJ problem is powerful in detecting false positive solutions and problems that contain false positive solutions.

Number and ratio of false positive solutions: Table 5 shows the number and ratio of false positive solutions as well as the problems with false positive solutions. We find 954 Python false positive solutions over 115 problems, 796 Java false positive solutions over 158 problems, and 2,040 C++ false positive solutions over 329 problems.

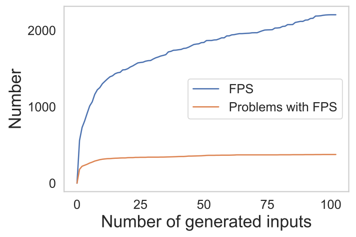


Figure 8: RQ2: The effect of the number of generated inputs. We find that 100 inputs per problem are adequate enough in finding the problems with false positive solutions.

Table 5: RQ2: Number of false positive solutions and number of OJ problems with false positive solutions. Overall, 43.1% of problems are found to contain false positive solutions.

Language	False positive solutions (FPS)	Problems with FPS
Python	954 (0.9%)	115 (12.9%)
Java	796 (0.8%)	158 (17.7%)
C++	2,040 (1.4%)	329 (36.9%)
Total	3,700 (1.1%)	384 (43.1%)

Interestingly, from the last row of Table 5, we observe that as many as 43.1% of the OJ problems are found to have false positive solutions. This high ratio is out of our expectations, especially considering the popularity and the large number of users of the OJ platform we study herein, and the extremely good coverage and mutation score of the test suites we observed in RQ1. This brings a question regarding whether coverage and mutation score is effective in test assessment for OJ platforms, which we further explore in RQ3.

Distribution of majority output ratio among the false positive solutions: When we feed a generated test input into different solutions for an identical problem, false positive solutions will have

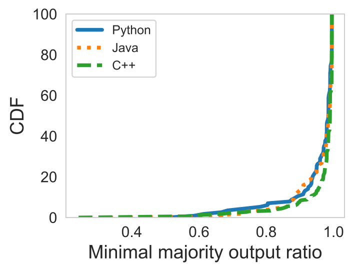


Figure 9: CDF of the minimal majority output ratio of problems with false positive solutions, 84.1% of the problems achieve a minimal majority output ratio larger than 0.9.

different test outputs from the majority of solutions. We thus further investigate the distribution of the output diversity on the false positive solutions we have detected. Specifically, we call the percentage of solutions with the majority output among all solutions of a problem as **majority output ratio**. For each OJ problem, every generated test input that leads to different outputs has a majority output ratio, and we present the smallest majority output ratio among all such inputs.

Figure 9 shows the distribution. We can see that almost all the problems' minimum majority output ratio is larger than 0.5. A larger majority output ratio often indicates more reliable bug detection results. There are 14 problems whose outputs are extremely diverse, and whose majority output ratio is below 0.5. The two first authors manually check the correctness of their solutions to guarantee that the solutions with majority outputs are indeed correct, and thus the solutions with different outputs are buggy. More characteristics of the false positive solutions are discussed in Section 5.1.

Answer to RQ2: The differential testing method we adopt detects 3,700 bugs, which are ignored by the existing OJ tests. A large proportion (43.1%) of the OJ problems we study are found to have false positive solutions.

4.3 RQ3: Effectiveness of Test Suite Assessment

For each solution, both coverage and mutation score can provide assessment results related to the reliability of the solution's test suite. A high assessment score is expected to provide developers with confidence that the solution is less likely to be buggy. To answer RQ3, we analyze the number of false positive solutions with full coverage and mutation score to see how many bugs would be ignored if developers choose to trust the assessment results provided by the coverage and mutation score values. Table 6 shows the results. In each cell, the larger the number and ratio, the less effective the metric is in assessing test reliability.

The first observation we immediately notice is the surprisingly high ratios of false positive solutions with a full line coverage. In particular, as high as 97.7% of the false positive solutions in Python and 96.3% of the false positive solutions in C++ have 100% line coverage. This means that the test suites for those solutions are assessed as perfect by line coverage, but they fail to detect the bug(s) in the solutions, yielding false positive solutions. In other words, for Python and C++ solutions, the line coverage hardly provides any useful information in judging whether the test suite is adequate in revealing bugs. If developers choose to trust the assessment results of line coverage, the majority of false positive solutions have perfect line coverage and will be undetected.

Table 6: Number and ratio of false positive solutions with perfect coverage and mutation score. Totally, the test suites achieve full line coverage, full branch coverage, full mutation scores for 88.7%, 79.0% and 37.8% of the false positive solutions, respectively.

Language	Line Coverage	Branch Coverage	Mutation Score
Python	389 (97.7%)	382 (96.0%)	239 (60.2%)
Java	331 (69.5%)	284 (59.7%)	138 (28.9%)
C++	701 (96.3%)	600 (82.4%)	229 (31.5%)
Total	1,421 (88.7%)	1,266 (79.0%)	606 (37.8%)

When comparing line coverage, branch coverage, and mutation score, we observe that for all three programming languages, there are lower ratios of false positive solutions with a full branch coverage than with a full line coverage. These observations suggest that branch coverage is more powerful than line coverage in exposing the weakness of tests. The ratios of mutation scores are the lowest among the three assessment metrics. However, considering the prevalence of equivalent mutants, it is difficult to reach a conclusion about the effectiveness of mutation score in avoiding false positive solutions.

When comparing different programming languages, the three metrics are the most effective when assessing Java solutions, and the least effective when assessing Python solutions. We suspect that this is related to the fact that the Java language is more redundant than other languages: inputting and outputting, creating data structures, etc. in Java requires writing more complex code. As a result, Java coders are more likely to copy a piece of template code to their submitted solution.

Answer to RQ3: Among the false positive solutions we have detected, 88.7% of them have full line coverage, 79.0% have full branch coverage, and 37.8% have full mutation scores. These observations reveal that a large ratio of bugs will be ignored if developers choose to fully trust the test assessment results provided by the coverage and mutation score of the existing OJ test suites.

5 DISCUSSION

This section discusses the characteristics of our detected false positive solutions, the threats to the validity of our study and the implications of our observations.

5.1 Characteristics of False Positive Solutions

This section digs deep into the characteristics of the false positive solutions we reported in RQ2. We randomly select 50 problems that

Table 7: Categories of false positive solutions. The examples in the last column can be found in our dataset [37].

Bug Type (Percentage)	Description	Example(Contest-Solution)
Corner case error (73.18%)	The algorithm does not consider corner case inputs.	ABC144B-147.py
Incorrect loop range (13.13%)	The upper or lower bounds of the loop statement are incorrect.	ABC135B-130.py
Incorrect type (5.03%)	The code does not convert types for variables.	ABC129A-118.py
Incorrect assignment (1.68%)	The assignment of variables is incorrect.	ABC157C-130.py
Error on float (4.47%)	The bug is caused by floating point precision errors.	ABC169C-104.py
Error on spelling (1.68%)	The bug is caused by the misspelling of variable names.	AGC014A-29.py
Hack solution (0.84%)	The code is not designed to solve the problem but to take advantage of the weakness of the tests to get passed.	ABC150A-65.py

contain false positive solutions and manually analyze their Python solutions. In total, 358 false positive Python solutions are analyzed.

As one might expect, investigating "the nature of false positive solution" is akin to investigating the (very wide) question of "the nature of software bugs". It is typically a highly context-sensitive issue and can necessitate a case-by-case answer.

Notwithstanding these inherent limitations, our manual analysis did highlight some similarities. At the solution level, we summarise the reasons for the bugs in false positive solutions in Table 7. In conclusion, most false positive solutions are buggy due to missing corner cases (i.e., their algorithms do not consider all possible cases and output incorrect answers on corner case inputs). There are also false positive solutions due to incorrect loop range, incorrect variable type, float errors, and incorrect spelling. Interestingly, we also find three hack solutions, which are not written to solve the problem but simply to hack the weak test cases with completely incorrect logic.

At the problem level, we find that different false positive solutions of the same problem often share identical bug reason due to the same corner case missed by the test suite. Specifically, 90% of the sampled problems contain bugs of only one type, the rest 10% of the problems contain bugs of two types and no problem contains bugs of more than two types.

A more comprehensive analysis would require substantial further work, which includes bug localisation and bug repair on each of the 3,700 false positive solutions and also a comparison with other bugs found in non-OJ systems. We have made our dataset public [37] to help fully facilitate such future work.

5.2 Threats to Validity

Dataset Construction. A potential threat to the external validity of our empirical study lies in the data we use. To mitigate this threat, we collect data from AtCoder, which is the only popular OJ platform that has open-sourced its full set of predefined tests available. We also choose problems published from 2016 to 2021 for data reliability. Additionally, the problems in our dataset come from different contests with different difficulties, which indicates a high diversity. Moreover, we extract all solutions from the CodeContests dataset, as these solutions have been pre-processed by clustering, filtering, and de-duplicating, to remove repeated solutions and invalid solutions [36, 55].

Test Oracle Reliability. Generating correct test oracles is the main challenge in our work. Differential testing can determine that there are some wrong programs giving wrong answers but cannot determine which answer is right, and it may introduce bias into our work. To mitigate this threat, we first ensure that all the problems we analyze with differential testing are not multi-answer problems by checking the description and the example input-output pair of every problem. Then we calculate the minimal majority answer ratio to get a better understanding of the validity of our test oracle as we can trust an answer with a very high percentage (more than 90%), and actually, we find more than 80% of such a high majority ratio. Moreover, two authors manually check the legality of generated inputs of all the problems that are found to have false positive solutions in differential testing.

Test Assessment Metrics and Tools. A possible threat to construct validity lies in the test assessment metrics and tools that we use. To mitigate this threat, we select the widely-used and well-studied [42] test metrics: line coverage, branch coverage, and mutation score to assess the reliability of the test suites of OJ problems. We also select the most popular open-source tools on GitHub [9, 46, 48] to carefully measure these metrics according to their documentation. Moreover, we configure the tools with the most common settings for the sake of generality [14, 53].

5.3 Implications and Challenges

Based on the preceding derived findings, we discuss implications and challenges for OJ platform developers, users, and researchers.

Implications for OJ platform developers. Our study reminds OJ platform developers that testing does not assure correctness. It is difficult to judge a program as bug-free even for widely accessed coding problems with strict input constraints (where the constraints free the programmers from considering the robustness of their solution programs). In addition, developers may risk test reliability if they fully trust the assessment results provided by the existing coverage and mutation score techniques. Furthermore, the results of RQ2 demonstrate that using a random test generation with differential testing can help improve the quality of test suites and cover more corner cases. OJ platform developers can also choose to update the test suite of problems routinely as the number of solutions grows with time, in this way, differential testing could be more powerful in finding corner cases as more versions of programs with the same function [19] become available.

Implications for OJ users. Online judge platforms have a huge user base. According to official reports, there are more than 150,000 registered users of AtCoder. The set of correct solutions is often made available to OJ platform users for them to learn and train how to code. Our study reveals that a passed solution is not always guaranteed to be a correct solution on the OJ platform. OJ platform users should be aware of this and not take for granted that the solutions that are accepted by the platforms are all correct solutions.

Implications for researchers. Based on our study, there are limitations of existing code coverage and mutation testing techniques in assessing the reliability of tests in bug detection. It is demanding for software engineering researchers to provide novel coverage and mutation testing techniques to combat the issues we find. For coverage criteria, our analysis on uncovered code (Section 4.1.1) indicates that new coverage criteria that disregard template code and dead code are needed; for mutation score, removing equivalent mutants can make mutation testing more practical. In this way, our results highlight the need for future work on equivalent mutant detection, especially for constraint-equivalent mutants.

The OJ problems and solutions can serve as well-defined datasets for software testing research purposes due to many of their unique characteristics. For example, researchers from the software engineering community can use such datasets for improving the effectiveness of bug prediction and detection techniques.

Moreover, recent years have witnessed the trend of using large language models for code generation and code completion. Most code generation tools (e.g., AlphaCode [36]) and commercial products (e.g., OpenAI's ChatGPT [49], GitHub's Copilot [1] and Tabnine [2]) adopt online judge problems and solutions in their training data. Our study reminds researchers from the artificial intelligence community of the existence of false positive solutions when they adopt datasets from online judge platforms. Researchers can choose to either filter the false positive solutions in the training data or conduct proper machine learning testing [72] and black-box repair [62, 63] to improve the correctness and robustness of the generated code.

6 RELATED WORK

This section summarises the related literature from two aspects: online judge systems and test suite reliability.

Online judge systems. OJ systems are firstly designed for education, and thus previous studies mainly focus on understanding the role that OJ platforms play in education, and on improving their educational effectiveness. Zhao et al. [77] observed that users tend to choose OJ problems in a sequential manner according to their positions or choose problems on the same topic. Based on this finding, they developed a Markov model to detect topics of OJ problems and recommend new problems to the users. Xia et al. [68] developed an interactive visualization system that recommends a learning path for OJ platform users based on other users' choices. Huang et al. [25] proposed a deep reinforcement learning framework to recommend exercises for users by jointly optimizing multi-objective functions such as smoothness of difficulty and engagement. Zhu et al.[78] developed a system namely HomoTR to recommend tests for a program solution by measuring the homology of two solutions.

Recently there are also studies focusing on automatic feedback and repair for programming solutions. These studies primarily propose feedback and repair techniques based on code clustering and code similarity using a large corpus of submitted solutions on the OJ platform [10]. Radicek et al. [56] proposed a program repair algorithm based on program matching. Perry et al. [51] proposed a program clustering algorithm based on quantitative semantic features for introductory programs such as OJ solutions. Song et al. [60] leveraged context information of OJ solutions to implement a function-level matching algorithm to automatically repair incorrect solutions. Tan et al. [64] collected incorrect OJ solutions and generated patches for them to build a benchmark for automated program repair techniques. All these studies rely on testing to judge the correctness of a solution, while our work reveals that the test suite might not be reliable.

Most recently, the OJ problem statement and the corresponding coding solutions have also been used as training data for big language models, such as AlphaCode [36] and CodeX [11].

Test assessment. The reliability (or effectiveness) of a test suite is a traditional and important topic in software testing. Code coverage is a general test suite reliability metric, the history of code coverage dates back to 1963 [43]. Software engineers have proposed different types of coverage criteria such as line coverage, branch coverage, path coverage, and modified condition/decision coverage [6, 54, 61]. Mutation testing is another widely-used metric for measuring the reliability of a test suite. Mutation testing was proposed in 1978 [16], and there has been a lot of research on mutation testing. The most recent work focuses on the empirical assessment of practical mutation testing [7, 52, 58] or improving the effectiveness and performance of mutation testing [26, 33]. In terms of the assessment of OJ tests, Li et al. [36] produced AlphaCode to automatically generate OJ solutions. They did a study to manually checked a sample of their generated solutions. The false positive solution rate is around 4%. However, they did not study the test assessment reliability on human solutions.

7 CONCLUSION & FUTURE WORK

In this paper, we presented a large-scale empirical study on the reliability of the test suites for 939 OJ problems with 541,552 user-submitted code solutions. We measured line coverage, branch coverage, and mutation scores for these solutions. Furthermore, we used differential testing to unveil the prevalence of false positive solutions. We found that 37,00 solutions are buggy yet they pass the test suite. Such solutions cover 43.1% of the problems available in the OJ platform under study. We have also revealed the limitation of using code coverage and mutation score to assess the reliability of the test suite for OJ solutions.

Our study took the first step in investigating the test suite reliability of online judge platforms. There are several promising future research directions. As online judge platforms are a wealthy source of programming problems and code for large language models, and are daily used by millions of users to improve and test their coding skills, it is worthwhile to explore the hidden values and threats behind their use.

ACKNOWLEDGEMENTS

This work is supported by the Beijing Outstanding Young Scientist Program under the grant number BJJWZYJH01201910001004. Zhenpeng Chen, Federica Sarro, and Mark Harman are supported by the ERC Advanced Grant under the grant number 741278 (EPIC: Evolutionary Program Improvement Collaborators).

The data used in this paper was curated and analyzed by the first two authors. This work did not involve any web content crawling processes; all data used in the paper comes from existing previously publicly shared data sources.

REFERENCES

[1] [n. d.]. Github Copilot. https://github.com/features/copilot.

[2] [n. d.]. Tabnine. https://www.tabnine.com/.

[3] 2015. LeetCode - The world's leading online programming learning platform. https://leetcode.com

[4] 2022. Top website designers, developers, freelancers for your next project. https://topcoder.com/

[5] James H Andrews, Lionel C Briand, Yvan Labiche, and Akbar Siami Namin. 2006. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering* 32, 8 (2006), 608–624.

[6] Luciano Baresi and Mauro Pezzè. 2006. An Introduction to Software Testing. *Electron. Notes Theor. Comput. Sci.* 148, 1 (2006), 89–111. https://doi.org/10.1016/j.entcs.2005.12.014

[7] Moritz Beller, Chu-Pan Wong, Johannes Bader, Andrew Scott, Mateusz Machalica, Satish Chandra, and Erik Meijer. 2021. What It Would Take to Use Mutation Testing in Industry - A Study at Facebook. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021, Madrid, Spain, May 25-28, 2021.* IEEE, 268–277. https://doi.org/10.1109/ICSE-SEIP52600.2021.00036

[8] Jean Luca Bez, Neilor A Tonin, and Paulo R Rodegheri. 2014. URI Online Judge Academic: A tool for algorithms and programming classes. In *2014 9th International Conference on Computer Science & Education.* IEEE, 149–152.

[9] boxed. 2020. mutmut. https://github.com/boxed/mutmut.

[10] Anderson Pinheiro Cavalcanti, Arthur Barbosa, Ruan Carvalho, Fred Freitas, Yi-Shan Tsai, Dragan Gašević, and Rafael Ferreira Mello. 2021. Automatic feedback in online learning environments: A systematic literature review. *Computers and Education: Artificial Intelligence* 2 (2021), 100027.

[11] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, et al. 2021. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374* (2021).

[12] Zhenpeng Chen, Yanbin Cao, Yuanqiang Liu, Haoyu Wang, Tao Xie, and Xuanzhe Liu. 2020. A comprehensive study on challenges in deploying deep learning based software. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2020.* 750–762.

[13] 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 (ISSTA 2016).* Association for Computing Machinery, New York, NY, USA, 449–452. https://doi.org/10.1145/2931037.2948707

[14] Mull contributors. 2022. Supported Mutation Operators. https://mull.readthedocs.io/en/latest/SupportedMutations.html [Online; accessed November-2022].

[15] Fabiano Cutigi Ferrari, Alessandro Viola Pizzoleto, and Jeff Offutt. 2018. A Systematic Review of Cost Reduction Techniques for Mutation Testing: Preliminary Results. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW).* 1–10. https://doi.org/10.1109/ICSTW.2018.00021

[16] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer* 11, 4 (1978), 34–41. https://doi.org/10.1109/C-M.1978.218136

[17] 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

[18] Yu Dong, Jingyang Hou, and Xuesong Lu. 2020. An intelligent online judge system for programming training. In *International Conference on Database Systems for Advanced Applications.* Springer, 785–789.

[19] Robert B Evans and Alberto Savoia. 2007. Differential testing: a new approach to change detection. In *The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering: Companion Papers.* 549–552.

[20] Michal Forišek et al. 2006. On the suitability of programming tasks for automated evaluation. *Informatics in Education-An International Journal* 5, 1 (2006), 63–76.

[21] Rodrigo Elias Francisco and Ana Paula Ambrosio. 2015. Mining an Online Judge System to Support Introductory Computer Programming Teaching.. In *EDM (Workshops).*

[22] Ali Ghanbari and Andrian Marcus. 2021. Toward Speeding up Mutation Analysis by Memoizing Expensive Methods. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER).* 71–75. https://doi.org/10.1109/ICSE-NIER52604.2021.00023

[23] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. *arXiv preprint arXiv:2203.03850* (2022).

[24] Farah Hariri, August Shi, Vimuth Fernando, Suleman Mahmood, and Darko Marinov. 2019. Comparing Mutation Testing at the Levels of Source Code and Compiler Intermediate Representation. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST).* 114–124. https://doi.org/10.1109/ICST.2019.00021

[25] Zhenya Huang, Qi Liu, Chengxiang Zhai, Yu Yin, Enhong Chen, Weibo Gao, and Guoping Hu. 2019. Exploring multi-objective exercise recommendations in online education systems. In *Proceedings of the 28th ACM International Conference on Information and Knowledge Management.* 1261–1270.

[26] Nargiz Humbatova, Gunel Jahangirova, and Paolo Tonella. 2021. DeepCrime: mutation testing of deep learning systems based on real faults. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021,* Cristian Cadar and Xiangyu Zhang (Eds.). ACM, 67–78. https://doi.org/10.1145/3460319.3464825

[27] AtCoder Inc. 2012. AtCoder. https://atcoder.jp

[28] AtCoder Inc. 2016. Atcoder testcases. https://atcoder.jp/posts/21

[29] 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.* 435–445.

[30] jacoco. 2022. jacoco. https://github.com/jacoco/jacoco.

[31] JaroslavHavrda. [n. d.]. Mull corrupted LLVM module · issue #1004 · mull-project/mull. https://github.com/mull-project/mull/issues/1004

[32] Yue Jia and Mark Harman. 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering* 37, 5 (2010), 649–678.

[33] Samuel J. Kaufman, Ryan Featherman, Justin Alvin, Bob Kurtz, Paul Ammann, and René Just. 2022. Prioritizing Mutants to Guide Mutation Testing. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022.* ACM, 1743–1754. https://doi.org/10.1145/3510003.3510187

[34] Adrian Kosowski, Michał Małafiejski, and Tomasz Noiński. 2007. Application of an online judge & contester system in academic tuition. In *International Conference on Web-Based Learning.* Springer, 343–354.

[35] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis and Transformation. San Jose, CA, USA, 75–88.

[36] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-Level Code Generation with AlphaCode. *arXiv preprint arXiv:2203.07814* (2022).

[37] Kaibo Liu, Yudong Han, Jie M. Zhang, Zhenpeng Chen, Federica Sarro, Mark Harman, Gang Huang, and Yun Ma. 2023. Replication package for this paper. https://github.com/maghsk/false-positive-solutions-in-competitive-programming

[38] Yiling Lou, Zhenpeng Chen, Yanbin Cao, Dan Hao, and Lu Zhang. 2020. Understanding build issue resolution in practice: symptoms and fix patterns. In *Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2020.* 617–628.

[39] luogu dev. 2019. Cyaron. https://github.com/luogu-dev/cyaron.

[40] Wei Ma, Mengjie Zhao, Ezekiel Soremekun, Qiang Hu, Jie M Zhang, Mike Papadakis, Maxime Cordy, Xiaofei Xie, and Yves Le Traon. 2022. GraphCode2Vec: generic code embedding via lexical and program dependence analyses. In *Proceedings of the 19th International Conference on Mining Software Repositories.* 524–536.

[41] William M. McKeeman. 1998. Differential Testing for Software. *Digit. Tech. J.* 10, 1 (1998), 100–107. http://www.hpl.hp.com/hpjournal/dtj/vol10num1/vol10num1art9.pdf

[42] Raymond McLeod Jr and Gerald D Everett. 2007. *Software Testing: Testing Across the Entire Software Development Life Cycle* (1 ed.). John Wiley, Hoboken.

[43] Joan C. Miller and Clifford J. Maloney. 1963. Systematic mistake analysis of digital computer programs. *Commun. ACM* 6, 2 (1963), 58–63. https://doi.org/10.1145/366246.366248

[44] Mikhail Mirzayanov. 2009. Codeforces. https://codeforces.com/

[45] Mikhail Mirzayanov. 2011. Codeforces contest rules. https://codeforces.com/blog/entry/4088

[46] Mull-project. 2022. mull. https://github.com/mull-project/mull.

[47] Kenko Nakamura. 2015. Atcoder problems models. https://kenkoooo.com/atcoder/resources/problem-models.json

[48] Nedbat. 2022. coveragepy. https://github.com/nedbat/coveragepy.

[49] OpenAI. 2022. ChatGPT. https://chat.openai.com/

[50] Mike Papadakis, Marinos Kintis, Jie M. Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation testing advances: an analysis and survey. In *Advances in Computers.* Vol. 112. Elsevier, 275–378.

[51] David M Perry, Dohyeong Kim, Roopsha Samanta, and Xiangyu Zhang. 2019. SemCluster: clustering of imperative programming assignments based on quantitative semantic features. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 860–873.

[52] Goran Petrovic, Marko Ivankovic, Gordon Fraser, and René Just. 2021. Does mutation testing improve testing practices?. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021.* IEEE, 910–921. https://doi.org/10.1109/ICSE43902.2021.00087

[53] PITest contributors. 2022. Mutation operators. https://pitest.org/quickstart/mutators [Online; accessed November-2022].

[54] Christian R. Prause, Jürgen Werner, Kay Hornig, Sascha Bosecker, and Marco Kuhrmann. 2017. Is 100% Test Coverage a Reasonable Requirement? Lessons Learned from a Space Software Project. In *Product-Focused Software Process Improvement - 18th International Conference, PROFES 2017, Innsbruck, Austria, November 29 - December 1, 2017, Proceedings (Lecture Notes in Computer Science, Vol. 10611)*, Michael Felderer, Daniel Méndez Fernández, Burak Turhan, Marcos Kalinowski, Federica Sarro, and Dietmar Winkler (Eds.). Springer, 351–367. https://doi.org/10.1007/978-3-319-69926-4_25

[55] Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladmir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. 2021. Project codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655* 1035 (2021).

[56] Ivan Radicek, Sumit Gulwani, and Florian Zuleger. 2018. Automated clustering and program repair for introductory programming assignments. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM.

[57] Miguel A Revilla, Shahriar Manzoor, and Rujia Liu. 2008. Competitive learning in informatics: The UVa online judge experience. *Olympiads in Informatics* 2, 10 (2008), 131–148.

[58] Ana Belén Sánchez, Pedro Delgado-Pérez, Inmaculada Medina-Bulo, and Sergio Segura. 2022. Mutation testing in the wild: findings from GitHub. *Empir. Softw. Eng.* 27, 6 (2022), 132. https://doi.org/10.1007/s10664-022-10177-8

[59] Patrick Schober, Christa Boer, and Lothar A Schwarte. 2018. Correlation coefficients: appropriate use and interpretation. *Anesthesia & Analgesia* 126, 5 (2018), 1763–1768.

[60] Dowon Song, Woosuk Lee, and Hakjoo Oh. 2021. Context-aware and data-driven feedback generation for programming assignments. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 328–340.

[61] Ting Su, Ke Wu, Weikai Miao, Geguang Pu, Jifeng He, Yuting Chen, and Zhendong Su. 2017. A Survey on Data-Flow Testing. *ACM Comput. Surv.* 50, 1 (2017), 5:1–5:35. https://doi.org/10.1145/3020266

[62] Zeyu Sun, Jie M. Zhang, Mark Harman, Mike Papadakis, and Lu Zhang. 2020. Automatic testing and improvement of machine translation. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 974–985.

[63] Zeyu Sun, Jie M Zhang, Yingfei Xiong, Mark Harman, Mike Papadakis, and Lu Zhang. 2022. Improving machine translation systems via isotopic replacement. In *Proceedings of the 44th International Conference on Software Engineering*. 1181–1192.

[64] Shin Hwei Tan, Jooyong Yi, Sergey Mechtaev, Abhik Roychoudhury, et al. 2017. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 180–182.

[65] Oscar Luis Vera-Pérez, Martin Monperrus, and Benoit Baudry. 2018. Descartes: A PITest Engine to Detect Pseudo-Tested Methods: Tool Demonstration. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. Association for Computing Machinery, New York, NY, USA, 908–911. https://doi.org/10.1145/3238147.3240474

[66] Szymon Wasik, Maciej Antczak, Jan Badura, Artur Laskowski, and Tomasz Sternal. 2018. A survey on online judge systems and their applications. *ACM Computing Surveys (CSUR)* 51, 1 (2018), 1–34.

[67] Huiting Wu, Yanshen Liu, Lin Qiu, and Yi Liu. 2016. Online judge system and its applications in c language teaching. In *2016 International Symposium on Educational Technology (ISET)*. IEEE, 57–60.

[68] Meng Xia, Mingfei Sun, Huan Wei, Qing Chen, Yong Wang, Lei Shi, Huamin Qu, and Xiaojuan Ma. 2019. Peerlens: Peer-inspired interactive learning path planning in online question pool. In *Proceedings of the 2019 CHI conference on human factors in computing systems*. 1–12.

[69] Qian Yang, J Jenny Li, and David Weiss. 2006. A survey of coverage based testing tools. In *Proceedings of the 2006 international workshop on Automation of software test*. 99–103.

[70] Fuchen Ying, Pengcheng Xu, and Di Xie. 2003. Welcome to PKU Judgeonline. http://poj.org/

[71] Tingting Yu, Zunchen Huang, and Chao Wang. 2020. ConTesa: Directed Test Suite Augmentation for Concurrent Software. *IEEE Transactions on Software Engineering* 46, 4 (2020), 405–419. https://doi.org/10.1109/TSE.2018.2861392

[72] Jie M. Zhang, Mark Harman, Lei Ma, and Yang Liu. 2020. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering* 48, 1 (2020), 1–36.

[73] Jie M. Zhang, Feng Li, Dan Hao, Meng Wang, Hao Tang, Lu Zhang, and Mark Harman. 2019. A study of bug resolution characteristics in popular programming languages. *IEEE Transactions on Software Engineering* 47, 12 (2019), 2684–2697.

[74] Jie M. Zhang, Lingming Zhang, Dan Hao, Meng Wang, and Lu Zhang. 2019. Do pseudo test suites lead to inflated correlation in measuring test effectiveness?. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 252–263.

[75] Tianyi Zhang, Cuiyun Gao, Lei Ma, Michael R. Lyu, and Miryung Kim. 2019. An Empirical Study of Common Challenges in Developing Deep Learning Applications. In *Proceedings of the 30th IEEE International Symposium on Software Reliability Engineering, ISSRE 2019*. 104–115.

[76] Yucheng Zhang and Ali Mesbah. 2015. Assertions are strongly correlated with test suite effectiveness. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 214–224.

[77] Wayne Xin Zhao, Wenhui Zhang, Yulan He, Xing Xie, and Ji-Rong Wen. 2018. Automatically learning topics and difficulty levels of problems in online judge systems. *ACM Transactions on Information Systems (TOIS)* 36, 3 (2018), 1–33.

[78] Chenqian Zhu, Weisong Sun, Qin Liu, Yangyang Yuan, Chunrong Fang, and Yong Huang. 2020. HomoTR: online test recommendation system based on homologous code matching. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1302–1306.

[79] IS Zinovieva, VO Artemchuk, Anna V Iatsyshyn, OO Popov, VO Kovach, Andrii V Iatsyshyn, YO Romanenko, and OV Radchenko. 2021. The use of online coding platforms as additional distance tools in programming education. In *Journal of Physics: Conference Series*, Vol. 1840. IOP Publishing, 012029.