On the Impact of Flaky Tests in Automated Program Repair

Yihao Qin*, Shangwen Wang*, Kui Liu†‡, Xiaoguang Mao*, Tegawendé F. Bissyandé§

*National University of Defense Technology, Changsha, China, {yihaoqin, wangshangwen13, xgmao}@nudt.edu.cn
†State Key Laboratory of Mathematical Engineering and Advanced Computing, Wuxi, China
‡University of Luxembourg, Luxembourg, tegawende.bissyande@uni.lu

Abstract—The literature of Automated Program Repair is largely dominated by approaches that leverage test suites not only to expose bugs but also to validate the generated patches. Unfortunately, beyond the widely-discussed concern that test suites are an imperfect oracle because they can be incomplete, they can include tests that are flaky. A flaky test is one that can be passed or failed by a program in a non-deterministic way. Such tests are generally carefully removed from the repair benchmarks. In practice, however, flaky tests are available test suite of software repositories. To the best of our knowledge, no study has discussed this threat to validity for evaluation of program repair. In this work, we highlight this threat and further investigate the impact of flaky tests by reverting their removal from the Defects4J benchmark. Our study aims to characterize the impact of flaky tests for localizing bugs and the eventual influence on the repair performance. Among other insights, we find that (1) although flaky tests are few (≈0.3%) of total tests, they affect experiments related to a large proportion (98.9%) of Defects4J real-world faults; (2) most flaky tests (98%) actually provide deterministic results under specific environment configurations (with the jdk version influencing the results); (3) flaky tests drastically hinder the effectiveness of spectrum-based fault localization (e.g., the rankings of 90 bugs drop down while none of the bugs obtains better location results compared with results achieved without flaky tests); and (4) the reparability of APR tools is greatly affected by the presence of flaky tests (e.g., 10 state of the art APR tools can now fix significantly fewer bugs than when the benchmark is manually curated to remove flaky tests). Given that the detection of flaky tests is still nascent, we call for the program repair community to reconsider APR results when flaky tests therefore impact performance evaluation of APR?

The common (and implicitly-accepted) assumption in the literature of test-based program repair is thus the execution of all test cases yield deterministic results [9]. The reality however is that flaky tests exist. Simply put, these are tests that sometimes fail, but that may pass if you give it enough try [10]. Such non-determinism (i.e., random results on the same configuration [11]) significantly hinders continuous integration activities and regression testing campaigns in the real-world. So far, several companies such as Google [12] and Facebook [13] have reported the influences brought by flaky tests. According to the statistics, around 1.5% tests in Google [12] and 4.6% in Microsoft [13] are flaky, which represent a non-negligible effort in their test activities.

The ultimate goal of APR is to be applied in practice to help programmers reduce the debugging burden. With SapFix [14], an industry giant like Facebook is exploring in industrial setting how automatic fix suggestions can be useful. In the academic literature, researchers evaluate the performance of their approaches by relying on various defect benchmarks that are built via mining real-world repositories for bugs and associated test cases. Defects4J [15] and BEARS [16] are major representatives of such benchmarks. While some recent approaches (e.g., iFixR [17]) investigate bug reports as a substitute to potentially-unavailable test suites, it is noteworthy that a large majority of test-based APR heavily rely on test cases within defects benchmarks. How do flaky tests therefore impact performance evaluation of APR? To the best of our knowledge, this question has not been studied in the literature.

In the Defects4J [15] benchmark, flaky tests have been manually removed from the test suites that are associated to their original programs. This creates an artificial scenario that does not reflect the practical constraints under which an APR tool would be leveraged in development settings. Our objective in this work is thus to reconsider APR results when flaky tests cannot be manually identified. Our empirical study thus assesses the impact of flaky tests on the repair pipeline. We first find out all the flaky tests in the defect benchmark of APR.

I. INTRODUCTION

Recent achievements in Automated Program Repair (APR) [1], [2], [3] constitute significant milestones in software automation. One of the most successful paradigm in APR is “generate-and-validate”, where test-based approaches recurrently break records on the number of benchmark bugs that can be automatically repaired [4], [5], [6], [7]. Our work focuses on this research line in which an APR pipeline relies on developer-provided test suite to locate buggy program elements (e.g., statements) and then deploys various transformation strategies to generate patches, which will be considered valid only if, when applied to the program, they let it pass all previously failing test cases as well as all previously passing test cases (i.e., the regression tests [8]).

Authorized licensed use limited to: NANJING UNIVERSITY OF AERONAUTICS AND ASTRONAUTICS. Downloaded on September 30,2021 at 05:43:06 UTC from IEEE Xplore. Restrictions apply.

DOI 10.1109/SANER50967.2021.00035
295
2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)
and investigate the proportion of flaky tests within the real-world test suites. We then re-check their flakiness to see if the testing results are random, after which we assess the impact of these flaky tests on spectrum-based fault localization, a widely-used fault localization technique in APR pipeline. At last, we assess the impact of flaky tests on the repair performance of APR tools.

Specifically, to perform this study, we consider all the tests in Defects4J benchmark that have been annotated as flaky tests and where excluded from the benchmark. The relevant flaky test set includes 213 unique tests associated to 387 Defects4J bugs. Actually, when we consider the different program versions, our study is conducted on a total of 3,940 instances of flaky tests. We run each test in several rounds (10 times) to assess the randomness of their results. For the investigation about the effect of flaky tests on fault localization, we leveraged an off-the-shelf spectrum-based localizer (i.e., GZoltar [18] with Ochiai [19]). Finally, our study considers 11 APR tools (10 from the RepairThemAll framework [20] and the recent TBar [4] tool) for comparing repair patches generated based on fault localization information obtained with test suites including or excluding flaky tests. The impact of flaky tests on the patch validation is assessed a series of comparative trials with/without flaky tests. We mainly find that:

- Flaky tests are rare in real-world defect benchmarks (e.g., only 0.3% test cases associated to Defects4J bugs are flaky). However they can persist in the test suite for a long time (over 50% of flaky tests have been flagged only after over 300 days). Flaky tests are coexistence of a wide range of real-world bugs as well.
- Most flaky tests (≈98%) will actually lead to deterministic results under certain specific configurations. However, the JDK version can influence the results.
- Flaky tests can lead to a significant decrease in fault localization efficacy: when using jdk-1.7, compared with performance metrics obtained without flaky tests, localization results of 90 (out of 387) bugs are degraded; in no case we have seen flaky tests help to improve suspicious locations ranking.
- The repairability performance of APR tools on Defects4J benchmark is substantially impacted by flaky tests: the state of the art TBar tool is now able to generate correct patches for only 17 bugs, going down from the 42 that it fixed with a curated test suite.

These findings call for more concentrations on impacts brought by flaky tests in future studies for making APR techniques more practical. According to our experiment results, we further point out a potential way to alleviate the influences that is to pay attention to partial fixes.

II. BACKGROUND

A. Automated Program Repair

Automated Program Repair (APR) is an increasing hot topic in recent years [2], [21]. Given a test suite with passing and failing tests, APR tools first perform Fault Localization to rank the buggy program element (e.g., method or statement) at a top position. In the community, various approaches have contributed on the fault localization: spectrum-based techniques [22], [23], [24], [25], mutation analysis [26], [27], applying advanced deep learning techniques [28], [29], and even utilizing patch generation information for fault localization [30].

With a ranked list of buggy code elements, APR tools adopt various strategies for Patch Generation, which is conducting the code transformation to generate patch candidates. Such as the heuristics-based APR relies on different search strategies (e.g., genetic programming [31] and random search [32]), pattern-based APR leverage the pre-defined/manually-mined code transformation patterns [4], [33], [34], [35], [36]. Constraint solving [37], [38] and deep learning approaches [7], [39], [40] have been explored to synthesizing patches as well.

After generating patches, Patch Validation is the final step to validate the correctness of generated patches. Validating patches by passing all test suites cannot ensure the correctness of outputted patches since human-written test suites are not adequate [41]. This leads to the overfitting problem [42]: a patch can make the program pass all the tests, but does not fix its bug. Since then, researchers denote a patch that can pass all the original test suite as a plausible patch. If a plausible is indeed semantically equivalent to developer-provided patch, it is identified as correct patch, otherwise it is considered as overfitting. Over the years, many methods have been proposed to alleviate the overfitting problem [43], [44], [45] and recent studies demonstrate that this direction still deserves exploration [46], [47], [48].

The fault localization and patch validation highly rely on the quality of test cases in the buggy programs, which will further impact the bug-fixing performance of APR systems.

B. Flaky Tests

Flaky tests are tests that provide non-deterministic results: they can pass or fail even under identical code versions. Flaky tests are now considered as a pervasive and serious problem in the community [12], [13] with considerable recent studies aiming to understand their naturalness, detect them and mitigate them.

Reproducibility is a primary and difficult problem as reported by Luo et al. [10], Lam et al. [49] also evaluated the reproducibility of flaky tests and found that the likelihood to reproduce the failures of flaky tests on different projects can range between 17% to 43%. Eck et al. [50] interviewed 21 professional MOZILLA developers and their answers manifested that reproducing the flaky behavior is one of the major challenges. Another problem is that although researchers have found ignoring flaky-test failures can induce more crashes [51], there are still situations where developers choose to ignore flaky tests. Throve et al. [52] studied 77 commits in 29 Android projects that were relevant to flakiness and found there were 13% commits that simply skipped or removed the flaky tests. Lam et al. [49] examined the pull requests at
Microsoft using three datasets. Their study revealed that about 5% of the flaky tests were fixed by only removing the tests.

As the occurrence of flaky tests seems inevitable, efforts have been made to concentrate on combating flaky tests. Luo et al. [10] focused on common categorization of flaky-test fixes by identifying and analyzing the version-control commits. Their results show that the most common reasons of flakiness are \textit{async wait}, \textit{concurrency} and \textit{test order dependency}. Palomba and Zaidman [53] investigated 19,532 JUnit test methods belonging to 18 software systems and found that the refactoring of test smells induces the fixing of flaky tests. Based on their prior works, Lam et al. [49] studied the lifecycle of flaky tests and showed that categorization works on flaky tests also apply to proprietary projects.

Aiming at detecting flaky tests, PRADET [54] takes as input a test suite and a reference order, and collects dependency information of tests through dynamic data-flow analysis. PRADET further filters out all the unproblematic data dependencies with an iterative dependency refinement algorithm. Bell et al. [55] presented DeFlaker to detect flaky tests by utilizing lightweight differential coverage tracking to monitor the coverage of the latest code changes. Contemporary, Lam et al. [56] introduced a framework called iDFlakies to run test suite automatically according to user-defined configuration, to identify flaky tests associated with the related partial classification and the test order. Instead of dynamic approaches, Pinto et al. [57] evaluated the performance of five machine learning classifiers on detecting flaky tests and found them all performed well. The authors also extracted words which are apt to appear in flaky tests and aggregated them as the vocabulary of flaky tests.

Despite huge efforts have been made to alleviate the impacts from flaky tests, the re-occurrence and repair of flaky tests are still great challenges. In addition, the impact of flaky tests on some other fields which deeply depend on regression testing, such as program repair, has not ever been investigated yet. To the best of our knowledge, we are the first to focus on the impact of flaky tests on program repair.

III. STUDY DESIGN

A. Research Questions

In our study, we aim to investigate the following research questions.

- **RQ1.** What is the proportion of flaky tests within the real-world (uncurated) test suites associated to APR benchmarks? We first aim to understand the occurrence frequency of flaky tests in real-world defect benchmark. Specifically, we concentrate on the number of flaky tests and their lifespans (i.e., the duration between their occurrences and disappearance).

- **RQ2.** To what extent the results of the available flaky tests are indeed random? This question is to check whether flaky tests' executions will fail or pass randomly under specific environments.

- **RQ3.** What is the impact of flaky tests on the results of spectrum-based fault localization in program repair?

We try to investigate the influence of flaky tests on the fault localization for APR. Since the fault localization is the first step in APR pipeline, investigating the influence for the fault localization could help exploit the impact of flaky tests on program repair.

- **RQ4.** To what extent do flaky tests affect the performance of APR tools? Finally, we are going to assess the impacts of flaky tests on APR tools’ effectiveness on generating valid patches for the given buggy programs.

B. Subject Selection

We focus on Defects4J defect benchmark [15] since (1) all bugs in this benchmark are from real-world programs and are carefully curated and (2) this dataset is the most widely-used one in software testing tasks [21], [30], [46], [58], [59]. In our study, we use the Defects4j-V1.3.0 since this version has been widely used in the fault localization and APR research work.

We next introduce how we create our flaky tests dataset. Through our manual investigation, we observe that almost in each bug from Defects4J, some test methods are removed and annotated as \textit{flaky method}. Listing 1 gives an example in which a test method \texttt{testFindDomainBounds} within the test file \texttt{TimeSeriesCollectionTests.java} is silent. We also note that this phenomenon has been questioned before\footnote{https://github.com/rjust/defects4j/issues/355} and the developers of Defects4J claim that removing these tests is for ensuring that this framework always returns reliable results.

\begin{Verbatim}
Listing 1: A Silent Flaky Test in the Bug Chart-1.
\end{Verbatim}

We adopt a heuristic approach to collect the flaky tests. After checking out each bug, we use the string \textit{flaky method} for selection. Every test method that is mapped goes through another manual process to check whether the string is in the annotation. Finally, we select all test methods that are annotated as flaky by developers of Defects4J as the flaky tests dataset for this study.

IV. STUDY RESULTS

We now state the experimental results that are designed for the research questions in this paper.

A. RQ1: [Prevalence of Flaky Tests]

1) Experimental Object: We first dissect the naturalness of flaky tests with our flaky test dataset collected from the benchmark Defects4J. Please note that Defects4J framework provides a command \texttt{defects4j query} for reasoning about the metadata of a specific project. Hence, in our study, we can utilize this commend for obtaining detailed commit information of each bug.
2) Results: Note that in the benchmark Defects4J, a bug refers to a specific buggy project version in history. If there is at least one flaky test in the test suite of a buggy program, we label this bug as a flaky bug. Statistics on flaky bugs and flaky tests in Defects4J are shown in Table I.

<table>
<thead>
<tr>
<th>Project</th>
<th>Flaky bugs</th>
<th>#bugs</th>
<th>#flaky tests (unique)†</th>
<th>#tests†</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chart</td>
<td>1-26</td>
<td>26</td>
<td>305 (83)</td>
<td>60820</td>
</tr>
<tr>
<td>Closure</td>
<td>1-62,64-92,94-133</td>
<td>131</td>
<td>501 (31)</td>
<td>952347</td>
</tr>
<tr>
<td>Lang</td>
<td>1-3-65</td>
<td>64</td>
<td>864 (32)</td>
<td>119561</td>
</tr>
<tr>
<td>Math</td>
<td>1-102</td>
<td>106</td>
<td>457 (14)</td>
<td>258005</td>
</tr>
<tr>
<td>Mockito</td>
<td>1-38</td>
<td>38</td>
<td>1237 (74)</td>
<td>44067</td>
</tr>
<tr>
<td>Time</td>
<td>1-20,22-27</td>
<td>26</td>
<td>576 (25)</td>
<td>101361</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>387</td>
<td>391†</td>
<td>1536161</td>
</tr>
</tbody>
</table>

From the results, we find that nearly all bugs are suffering from flaky tests in that 98.9% (387/391) bugs are actually flaky bugs. Bugs from four projects (i.e., Chart, Lang, Mockito, and Time) all contain flaky tests in their test suites.

When looking at the numbers of flaky tests, we note the proportion of flaky tests compared with the whole tests is rather low (i.e., \( \approx 0.3\% \), 3 940/1 536 161). The project in which the flaky tests occur most frequently is Mockito whose percentage is around 3\% (1 237/44 067).

Finding-1: Although flaky tests are rare (only take account \( \approx 0.3\% \) of the number of total test methods), a large proportion of real-world programs in the Defects4J dataset (i.e., 98.9\%) is affected by flaky tests.

The previous results reveal that a small number of flaky tests can affect a large amount of bugs in the evolution process of software, which motivates us to further investigate the lifespan of each flaky test (i.e., the duration between its occurrence and disappearance). To calculate the lifespan value, we find out two timestamps according to the commit information for each flaky test. One is the first time the test occurs while another is the first time this test disappears after appearance. We consider the time between the two timestamps as the lifespan of a flaky test. Results are shown in Fig. 1.

![Fig. 1: The distribution of lifespan of flaky tests.](image)

Among the 258 unique flaky tests, less than 45\% (116/258) exist less than 100 days. As comparison, the lifespan of 112 flaky tests (43.4\%) exceeds 300 days which means it takes a long time for developers to remove the impacts of these tests. Moreover, the lifespan of a considerable proportion (i.e., 30.2\%, 78/258) of flaky tests is even larger than 1 000 days.

Finding-2: A large number of flaky tests (43.4\%) remain in program’s test suites for a long time (more than 300 days). During that time, they will keep affecting all testing activities.

B. RQ2: [Dissection of Flakiness]

1) Experimental Object: In this RQ, we investigate the randomness of flaky tests. To achieve so, we decide to run each of them consecutively for ten times under different environments. Previous study [10] introduces three platform-related factors which can affect the running results of flaky tests (i.e., operating system, jdk version, as well as browser version). Since our study subjects are not related with browser, we remove it from our experiment. For the other two factors, we choose Ubuntu-16.04 and 18.04 as variants for operating system and adopt widely-used jdk-1.7 and 1.8 as variants for jdk version. We thus perform this experiment under four environments which are composed by different combinations of operating system and jdk version.

2) Results: We define four categories of flaky tests here.

- **Pass** which denotes the test result is always passing in the ten times running;
- **Fail** which denotes the test result is always failing in the ten times running;
- **Random** which denotes the test result is sometimes passing while sometimes failing in the ten times running;
- **Timeout** which denotes the test result is not returned after a long-time waiting. This could happen because some flaky tests are caused by the Async Wait problem in which a threat may sleep for a long time [10].

Results are listed in Table II. We note that for most flaky tests, the running results are deterministic under specific environment settings. For instance, when performing under jdk-1.7 with Ubuntu-16.04, 98.7\% (3 890/3 940) flaky tests either pass for the ten times or fail for the ten times, while this figure becomes 98.8\% when the environment turns into jdk-1.8 with Ubuntu-18.04. Flaky tests from Chart project always fail no matter under what experiment setting.

Finding-3: A large amount of flaky tests (i.e., \( \approx 98\% \)) have deterministic results under specific environment settings. This finding is valid based on a series of 10 consecutive runs per experiment.

We also note that different environment variables contribute diversely to the randomness of testing results: the results can diverse a lot under different jdk versions while almost keeping the same under different operating system versions. Specifically, comparing the results from jdk-1.7 plus Ubuntu-16.04 with those from jdk-1.7 plus Ubuntu-18.04, the numbers
of \( P \) and \( F \) tests almost keep the same (2.598 vs. 2.529 and 1.292 vs. 1.257), the number of \( T \) tests is identical (i.e., 18), while the number of \( R \) tests is the only one that changes dramatically (i.e., from 32 to 136). From the perspective of Defects4J project, we note that only results of flaky tests from Mockito project change notably. To be detailed, the numbers of \( P \) and \( F \) tests both drop down mildly while the number of \( R \) tests increases from 28 to 129.

The trend becomes different when it comes to impacts of jdk version. Take the results from jdk-1.7 plus Ubuntu-16.04 and jdk-1.8 plus Ubuntu-16.04 as comparison. The number of \( P \) tests decreases sharply from 2.598 to 1.399 while the number of \( F \) tests increases significantly from 1.292 to 2.144. While the number of \( R \) tests is stable, the number of \( T \) tests skyrockets from 18 to 356. This tendency is similar when the operating system changes into Ubuntu-18.04 except that the number of \( R \) tests decreases while the number of \( T \) tests keep consistent. From the perspective of project, besides Mockito, notable differences can be found in other four projects which are Closure, Lang, Math, and Time. For instance, in the project Time, the number of \( P \) tests drops dramatically from 575 to 79 while the number of \( F \) tests climbs from 0 to 497.

<table>
<thead>
<tr>
<th>Project</th>
<th>NFT</th>
<th>P</th>
<th>F</th>
<th>R</th>
<th>T</th>
<th>P</th>
<th>F</th>
<th>R</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chart</td>
<td>305</td>
<td>0</td>
<td>305</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>305</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Closure</td>
<td>501</td>
<td>418</td>
<td>83</td>
<td>0</td>
<td>0</td>
<td>119</td>
<td>382</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Lang</td>
<td>864</td>
<td>456</td>
<td>407</td>
<td>1</td>
<td>0</td>
<td>429</td>
<td>434</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Mockito</td>
<td>1237</td>
<td>870</td>
<td>339</td>
<td>28</td>
<td>128</td>
<td>550</td>
<td>310</td>
<td>338</td>
<td>133</td>
</tr>
<tr>
<td>Time</td>
<td>575</td>
<td>575</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>79</td>
<td>497</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>3940</td>
<td>2598</td>
<td>1292</td>
<td>32</td>
<td>18</td>
<td>1399</td>
<td>2144</td>
<td>41</td>
<td>356</td>
</tr>
</tbody>
</table>

*\( \text{NFT} \) denotes number of flaky tests. \( P, F, R, T \) denote Pass, Fail, Random, and Timeout, respectively.*

Given that some flaky tests tend to have different results under different jdk versions, we decide to investigate their change trends (i.e., whether they tend to pass on a jdk version while fail on another version or vice versa). Note that we exclude the differences brought by operating system in the remained part of this paper in that our results reveal that these impacts are rather tiny. The results are shown in Table III.

We totally conclude ten patterns within which four indicate the testing result is consistent while the other six illustrate differences. We note that over 70\% (2765/3940) tests still possess consistent testing results under different jdk versions, among which the number of tests whose results are always passing is slightly larger than that of tests whose results are always failing (1497 vs. 1248).

When running results are changed, diverse situations may happen. The dominant change pattern is from passing to failing (i.e., \( PF \) in the table) which takes place \( \approx \)86\% (1005/1175) of the total number. We also note that the number of tests whose results alter from failing to passing (i.e., \( FP \) in the table) is rather small (only 8). Such a result shows that for flaky tests which demonstrate randomness across different jdk versions, they tend to pass on jdk-1.7 and fail on jdk-1.8 while the reverse direction is unpopular. There are also some instances in which \( R \) tests become \( P \) tests, \( F \) tests become \( R \) tests, or \( P \) tests change into \( R \) tests. Nonetheless, they all occur no more than 100 times.

Further manual investigation shows that the trend is similar when the operating system is Ubuntu-16.04. We thus only illustrate this table here due to space constraint.

**Finding-5** | For most of the flaky tests (i.e., over 70\%), their testing results appears to not be influenced by the jdk version. Those results change when tested on diverse jdk versions tend to pass on a lower version (i.e., 1.7) while fail on a higher version (1.8).

---

**C. RQ3: [Impacts on Fault Localization]**

We in this RQ investigate the impacts of flaky tests on the effectiveness of Fault Localization. From this RQ on, we only perform experiments on Ubuntu-18.04 since the impact of operating system is neglectable as we have shown.

1) **Experimental Object**: We choose to utilize a widely-used off-the-shelf fault localization framework, GZoltar [18] with the latest version (V1.7), for conducting this experiment. This FL tool has also been integrated into the pipeline of a large amount of APR tools [4], [5], [37], [60]. To perform this experiment, we first execute this tool without flaky tests under jdk-1.7 and 1.8, respectively and record the rankings of buggy lines. We then re-execute it with flaky tests under the same environment and make comparison between the results obtained with and without flaky tests for reporting our findings. Note that for bugs which possess multiple buggy lines, we consider their ranking results the same with the top-ranked buggy line by following previous study [30].

2) **Results**: We briefly recall that GZoltar is a spectrum-based fault localization (SBFL) tool which leverages runtime information for calculating suspiciousness for program elements (e.g., statements). In our study, suspiciousness is calculated by Ochiai algorithm [19] with the idea that a statement covered by more failing tests and less passing tests would be more likely to be buggy. Statements are ranked based on their suspiciousness according to a descending order, that is, a statement with suspicious value being 1 will always be
To better investigate the differences in the fault localization results with/without flaky tests, we provide the distribution of rankings for each locable bug (i.e., bugs that can always be located whatever the environment is) under diverse experimental settings (i.e., with/without flaky tests and the related JDK version). Results are shown in Figure 2, where white box denotes results obtained with flaky tests, black box denotes results obtained without flaky tests, and “Rank” represents the bug position in the ranked list of suspicious code elements reported by the fault localization tool. Overall, with the results (except locating Mockito bugs under JDK-1.8), we note that fault localization without flaky tests can rank bugs on higher positions than the fault localization with flaky tests. It indicates that flaky tests can impact the performance of fault localization for program repair. Moreover, this phenomenon is independent with the used JDK version. For instance, for project Chart, the medium values of the ranking results obtained with and without flaky tests are 7.5 and 3 respectively under both JDK versions.

The only exception happens in Mockito project under JDK-1.8 where the medium values of black and white boxes are identical while the upper quartile of the black box is larger than that of the white box, which means the localization results obtained with flaky tests are better than those obtained without flaky tests. Such a result is caused by the fact that there are only locable bugs in this project and the localization performance improvement of Mockito-10 is significant (i.e., the ranking of buggy statement rises 45 positions) while the performance degradation of other four bugs is insignificant (i.e., the ranking only drops 1, 1, 10, 12 positions respectively).

Another interesting phenomenon here is that for bugs from Time project, their localization results are completely the same whether flaky tests are included under JDK-1.7. This can be explained by results listed in Table II that under this configuration (i.e., JDK-1.7 & Ubuntu-18.04), nearly all of the flaky tests (573/576) in Time project can always pass, which can be explained by results listed in Table II that under this configuration (i.e., JDK-1.7 & Ubuntu-18.04), nearly all of the flaky tests (573/576) in Time project can always pass, which indicates the flaky tests may have limited influence. Mockito project, which contains a larger number of flaky tests than Time, also possesses the same trend. Our in-depth analysis finds that this is due to the technical problem occurred during the execution of GZoltar on this project: the flaky tests which should be failing are skipped by GZoltar. Previous question\(^2\) indicates that different JDK version does bring impacts on experimental results.

So far we have only investigated the overall situation of fault localization results. We have not yet dissected the changes in the FL results of each locable bug. To achieve so, we define

\(^2\)https://github.com/GZoltar/gzoltar/issues/6
TABLE IV: Fault localization results with flaky tests compared against those obtained without flaky tests.

<table>
<thead>
<tr>
<th>Project</th>
<th>#bugs</th>
<th>↑→ ↓</th>
<th>Not located</th>
<th>Timeout</th>
</tr>
</thead>
<tbody>
<tr>
<td>jdk1.7 + Ubuntu-18.04</td>
<td>jdk1.8 + Ubuntu-18.04</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chart</td>
<td>26</td>
<td>0</td>
<td>5 19</td>
<td>2</td>
</tr>
<tr>
<td>Closure</td>
<td>131</td>
<td>0</td>
<td>46 80</td>
<td>1</td>
</tr>
<tr>
<td>Lang</td>
<td>64</td>
<td>0</td>
<td>4 22</td>
<td>0</td>
</tr>
<tr>
<td>Math</td>
<td>102</td>
<td>0</td>
<td>41 5</td>
<td>0</td>
</tr>
<tr>
<td>Mockito</td>
<td>38</td>
<td>0</td>
<td>2 31</td>
<td>0</td>
</tr>
<tr>
<td>Time</td>
<td>26</td>
<td>0</td>
<td>22 3</td>
<td>0</td>
</tr>
<tr>
<td>Total</td>
<td>387</td>
<td>0</td>
<td>120 159</td>
<td>3</td>
</tr>
</tbody>
</table>

↑ denotes number of bugs whose location results are improved (i.e., rankings of buggy statements are higher), → denotes number of bugs whose location results are unchanged, ↓ denotes number of bugs whose location results are decreased (i.e., rankings of buggy statements are lower), not located denotes those cannot be located both with and without flaky tests. As shown in Table II, 18 tests are timeout and thus cannot obtain location results.

![Fault localization results under diverse experimental settings.](image)

Fig. 2: Fault localization results under diverse experimental settings.

Finding 7: Fault localization disturbances brought by flaky tests can generally be limited (i.e., less than 20) while sometimes be rather significant (i.e., larger than 100).

D. RQ4: [Impacts on Repair Performance]

In this RQ, we aim to investigate the impacts of flaky tests on the repairability of APR tools. We rerun diverse APR tools which are suggested to be executed under specific jdk versions.

![Distribution of disturbance degree after including flaky tests.](image)

Fig. 3: Distribution of disturbance degree after including flaky tests.

We thus do not distinguish differences brought by jdk versions in this RQ.

1) Experimental Object: We choose to use a recently-proposed framework, RepairThemAll [20], since it integrates a large number of APR tools (i.e., 11 in total). Besides, we also execute a state-of-the-art template-based APR tool, TBar [4], for a more comprehensive assessment. To perform this experiment, we rerun each tool under Ubuntu-18.04 by feeding them fault localization results obtained through last step. Specifically, we feed those whose recommended jdk version is 1.7 (i.e., tools from Arja system [60] and TBar) with results obtained under jdk-1.7 while feed those whose recommended version is 1.8 (i.e., tools from Astor system [61], Nopol [37], and DynaMoth [62]) with results obtained under jdk-1.8. Following a recent study [21], we stop each repair attempt if (1) the first plausible patch (i.e., a patch that can
pass all the tests) is generated or (2) the number of generated patch candidates (NPC) reach 10,000 but the tool still does not find a plausible patch. We exclude NPEFix [63] since it does not require a traditional fault localization step and thus the disturbance of fault localization results cannot influence it.

2) Results: Results of the 10 tools within RepairThemAll framework are listed in Table V. In this table, we adopt recall as a measurement for repair performance. The value is calculated as number of generated plausible patches divided by the total number of bugs. Results without flaky tests are from the original experiment of RepairThemAll [20] while the correctness of patches is manually labelled by Tian et al. [47]. Note that in this table we remove results on 4 duplicated bugs listed in Defects4J-V2.0. In our experiment, we also label the patch correctness according to the guidance provided by Liu et al. [21].

We note that all APR tools suffer from decreases of recall. For instance, without flaky tests, Nopol can generate totally 105 plausible patches while the number drops down to 14 when involving flaky tests. Consequently, the recall of Nopol decreases 23.51%, which is the largest degree among the 10 tools. For other tools, Arja also experiences a sharp decrease of 20.41% while Cardumen and jMutRepair only lose slightly (around 3%) which is mainly because they are unable to fix a large number of bugs originally. Further investigation shows that all bugs that are previously fixed but not fixed this time are due to the existence of flaky test which are always failing. Another obvious phenomenon is that APR tools can seldom correctly fix bugs when encountering flaky tests: only 6 correct patches are generated in our experiment while 6 tools cannot generate any correct patch at the same time. In comparison, every tool is capable of correctly fixing bugs and the total number of correct patches is 36 without flaky tests.

**Finding-8** Flaky tests tend to negatively affect the performance of APR tools. All 10 tools from RepairThemAll suffer from recall decreases, among which the most significant one can only plausibly fix 14 bugs while the original number is 107.

**Finding-9** The previously observed benchmark overfitting phenomenon does not hold when considering flaky tests in Defects4J.

We also rerun TBar [4] with flaky tests and demonstrate results in Table VII. We introduce the concept partial fix here which is previously proposed by Liu et al. [67]: a patch makes the buggy program pass a part of previously failed test cases without causing any new failed test cases is denoted as a partial fix.

From the results, we note TBar can completely fix only 34 bugs with flaky tests, less than half of the previous number which is 80. However, it can partially fix much more bugs this time (i.e., 51 vs. previous value 18). To better understand the relationship between fixed and partially fixed bugs, we investigate the overlap between them and demonstrate it in Figure 4.

A number of findings can be found. We first note a large proportion of previously fixed bugs (45%, 36/80) are now partially fixed, while another part of them (41.25%, 33/80) can still be fixed now. Manual validation reveals that different results are caused by the running results of the involved flaky tests: those whose flaky tests are failed can only be partially fixed while those whose flaky tests are passing can be fully fixed. Totally, more than 85% (85/98) bugs that can be fixed (both fully and partially) before can still be fixed now. We also note totally 13 (i.e., 11+2) bugs where patched can be generated before cannot be fixed now. We conclude the reasons from two aspects as the following: (1) 8 of them are due to the fact that the fault localization results decrease sharply so that TBar cannot generate a patch within the pre-defined NPC constraint (e.g., Chart-1 where the ranking of buggy statement drops 34 positions); (2) the left 5 (all in project Math) happen because the flaky tests are Timeout as shown in Table II (e.g., Math-85). It should be noted here that APR tools from RepairThemAll do not fail to fix bugs due to the compromised efficacy of fault localization. This can be explained by the fact that TBar integrates a large number of fix patterns so that it generates much more candidate patches at each suspicious location compared with other tools. Hence, it is much sensitive to the localization results as revealed by a previous study [21]. A special case is Math-20 which can be partially fixed before but completely fixed now. We find the reason is that the failed flaky tests help improve the ranking of a non-ground-truth location (i.e., statements not modified by developer-provided patch) to No.1 and TBar generates a patch there. Manual validation shows that this patch is plausible but incorrect (i.e., overfitting).

**Finding-10** With flaky tests involved, a large proportion (i.e., 45%) of previously fixed bugs are now partially fixed.
Authorized licensed use limited to: NANJING UNIVERSITY OF AERONAUTICS AND ASTRONAUTICS. Downloaded on September 30,2021 at 05:43:06 UTC from IEEE Xplore. Restrictions apply.

TABLE V: Repairability comparison of diverse APR tools on 5 benchmarks.

<table>
<thead>
<tr>
<th>Tool</th>
<th>RepairThemAll</th>
<th>Defects4J</th>
<th>intraClassJava</th>
<th>QuixBugs</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARJA</td>
<td>12 (2%)</td>
<td>21 (6%)</td>
<td>6 (1%)</td>
<td>23 (9%)</td>
<td>4 (10%)</td>
</tr>
<tr>
<td>jGenProg-A</td>
<td>1 (1%)</td>
<td>9 (1%)</td>
<td>3 (1%)</td>
<td>18 (6%)</td>
<td>4 (10%)</td>
</tr>
<tr>
<td>Kali-A</td>
<td>15 (5%)</td>
<td>24 (25%)</td>
<td>4 (1%)</td>
<td>5 (1%)</td>
<td>2 (5%)</td>
</tr>
<tr>
<td>RSRepair-A</td>
<td>1 (1%)</td>
<td>6 (&lt;1%)</td>
<td>7 (1%)</td>
<td>22 (7%)</td>
<td>4 (10%)</td>
</tr>
<tr>
<td>Cardumen</td>
<td>13 (5%)</td>
<td>12 (1%)</td>
<td>3 (1%)</td>
<td>0 (0%)</td>
<td>4 (10%)</td>
</tr>
<tr>
<td>jGenProg</td>
<td>13 (5%)</td>
<td>14 (1%)</td>
<td>7 (1%)</td>
<td>4 (1%)</td>
<td>3 (7%)</td>
</tr>
<tr>
<td>jKali</td>
<td>10 (5%)</td>
<td>8 (&lt;1%)</td>
<td>6 (1%)</td>
<td>5 (1%)</td>
<td>2 (5%)</td>
</tr>
<tr>
<td>jMutRepair</td>
<td>7 (2%)</td>
<td>11 (&lt;1%)</td>
<td>4 (1%)</td>
<td>24 (8%)</td>
<td>3 (7%)</td>
</tr>
<tr>
<td>Nopol</td>
<td>1 (1%)</td>
<td>72 (65%)</td>
<td>14 (3%)</td>
<td>32 (10%)</td>
<td>1 (2%)</td>
</tr>
<tr>
<td>DynaMoth</td>
<td>0 (0%)</td>
<td>124 (100%)</td>
<td>16 (4%)</td>
<td>6 (2%)</td>
<td>2 (5%)</td>
</tr>
</tbody>
</table>

TABLE VI: Repair performance of TBar with and without flaky tests.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Chart</th>
<th>Closure</th>
<th>Lang</th>
<th>Math</th>
<th>Project</th>
<th>Fixed</th>
<th>Partially fixed</th>
<th>Fixed</th>
<th>Partially fixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARJA</td>
<td>9/14</td>
<td>0/4</td>
<td>1/4</td>
<td>0/3</td>
<td>1/4</td>
<td>0/4</td>
<td>0/4</td>
<td>1/4</td>
<td>0/4</td>
</tr>
<tr>
<td>jMutRepair</td>
<td>7/11</td>
<td>1/5</td>
<td>6/8</td>
<td>1/8</td>
<td>6/8</td>
<td>1/8</td>
<td>1/8</td>
<td>6/8</td>
<td>1/8</td>
</tr>
<tr>
<td>Cardumen</td>
<td>5/14</td>
<td>0/3</td>
<td>1/4</td>
<td>3/8</td>
<td>1/4</td>
<td>3/8</td>
<td>1/4</td>
<td>3/8</td>
<td>1/4</td>
</tr>
<tr>
<td>Math</td>
<td>19/36</td>
<td>0/4</td>
<td>7/13</td>
<td>7/20</td>
<td>7/13</td>
<td>7/20</td>
<td>7/13</td>
<td>7/20</td>
<td>7/13</td>
</tr>
<tr>
<td>jGenProg-A</td>
<td>1/2</td>
<td>0/0</td>
<td>1/2</td>
<td>0/0</td>
<td>1/2</td>
<td>0/0</td>
<td>1/2</td>
<td>0/0</td>
<td>1/2</td>
</tr>
<tr>
<td>Nopol</td>
<td>1/3</td>
<td>1/2</td>
<td>1/3</td>
<td>1/2</td>
<td>1/3</td>
<td>1/2</td>
<td>1/3</td>
<td>1/2</td>
<td>1/3</td>
</tr>
</tbody>
</table>

V. DISCUSSION

A. Implications

Should program repair care flaky tests? The answer is obviously positive according to our findings: (1) flaky tests can exist in a large amount of real-world bugs, (2) flaky tests can negatively affect the fault localization results, and (3) APR tools’ performances can be greatly compromised when encountering flaky tests. Such results indicate that if APR techniques are going to be applied in practice, the impact brought by flaky tests is a big challenge that must be dealt with.

How to alleviate the impact from flaky tests? Given that there is currently no APR tools designed specially for addressing the problem brought by flaky tests, our results point out a way to alleviating the impact from flaky tests currently that is to pay attention to partial fix. Theoretically, a patch may be reasonable if it can pass the other previously failed tests whatever the result would be on the flaky test that would always fail. Our experimental results show the effectiveness of this strategy: if we pay attention to partial fixes, we can preserve 45% of the plausible patches. This inspires us that in practice, each patch that can partially fix a bug should be noticed since it may provide practical guidance for fixing the

B. Threats to Validity

Threats to external validity. Our external validity is mainly challenged by flaky test dataset we used: in this study, we only focus on flaky tests from six projects in Defects4J benchmark. Such tests may not represent in the wild flaky tests well and could either underestimate or overestimate the impacts of flaky tests on program repair. Nevertheless, this threat is mitigated considering that this benchmark is the most widely-used in software testing tasks. There are also a number of on-hand results on this benchmark from previous studies that we can make comparison with. Expanding the flaky tests scale and executing FL and APR tools on other bugs will increase the time-consuming and thus are considered as our future work. B. Threats to Validity

B. Threats to Validity

Figure 4: Overlaps in fixed and partially fixed Defects4J bugs of TBar.

3 https://github.com/rjust/defects4j/issues/340
is unrealistic for us to rerun each test. We thus consider this identification method as reasonable.

**Threats to construct validity.** For our experiments, we run flaky tests, FL tool (i.e., GZoltar-V1.7), as well as 11 APR tools under various configurations. It is certainly error-prone to conduct such a large-scale study, e.g., we have reported that due to the technical problem, fault localization results of Closure project are identical under jdk-1.7 no matter flaky tests are included or not. Such results may bring threats to our results and findings. We alleviate this via re-checking our experiment results for several times and proposing reasonable explanations for each phenomenon we observed. We also note that 10 times of running may not fully expose the flakiness of flaky tests as previous studies choose to execute 100 times [57]. It thus brings threats to the results illustrated in Table II. However, after randomly selecting 100 tests that have determined results and running consecutively 100 times under jdk-1.7 and Ubuntu-18.04, we find that their results are still consistent. Moreover, we release all the experimental data in this study for the convenience of the community to make further review.

**VI. RELATED WORK**

This empirical study focuses on the impact of flaky tests in Automated Program Repair. Results reveal that the ignorance of flaky tests may cause bias in the evaluation of performances of APR techniques. There are also some works in the literature that point out potential biases in APR research field.

Traditionally, a patch passing all the test cases was considered as correct. Long et al. [41] first questioned the rationality of this criterion since developer-provided tests are not adequate for using as specifications of program behaviours. Their investigation shows that patches passing all the tests can simply delete some statements or change the condition in an if statement to true or false, thus still being faulty. This is the well-known overfitting problem which is a direction in APR deserving more in-depth exploration [46], [47]. After that, APR tools tend to highlight precision (i.e., the proportion of real correct patches compared against all the generated ones) as a metric of their effectiveness [6], [68]. Nonetheless, how to determine the correctness of patch is still a challenging task. Some researchers manually checked the semantic equivalence between the generated patch and the ground-truth (i.e., developer-provided patch) [60], [69], while others used an independent test suite generated by test generation tools for assessment [70]. Le et al. [48] assessed the reliability of these two methods and found that a notable part of patches passing the independent test suite are still incorrect while manual validation can suffer from subjectivity. Wang et al. [71] further dissected the differences between machine-generated patches and ground-truth to provide guidance for future manual assessment. Liu et al. [21] concluded totally ten common code change patterns from correct patches for further easing this process.

As the first step in APR pipeline, the results of fault localization (FL) step can exert impacts on repair effectiveness. Liu et al. [67] first reported that current APR techniques are configured with diverse FL settings and thus it is biased to directly compare their repair performances. Their experiments illustrate that precise FL results can help boost the effectiveness of APR tools to a large extent, which is further confirmed by a recent large-scale empirical study [21] that demonstrates if given the ground-truth locations, APR will generate more correct patches and increase the efficiency at the same time compared with receiving location results from FL tools.

**Benchmark overfitting is another recently-observed phenomenon.** Wang et al. [72] found that in the community-adopted Defects4J, bugs from Mockito project are not complex than other bugs with respect to some features of the patches like lines of code, number of chunks, etc. However, state-of-the-art APR tools such as SimFix [5] and CapGen [6] tend to achieve poor performances on these bugs. Durieux et al. [20] assessed the repairability of 11 APR tools on 5 defect benchmarks. Their results indicate that APR tools tend to perform better on Defects4J (i.e., they can generate more patches that pass the test suite). Our study points out a reasonable explanation for this phenomenon that is Defects4J removes all the flaky tests.

A recent study [73] pointed out that an APR tool may fail to fix a bug due to some reasons not related to its repairability such as the incorrect operation from the performers of the experiment. This finding calls for more attention to establish unbiased evaluation of APR techniques.

**VII. CONCLUSION**

In this paper, we report on an exploratory study concerning the impacts of flaky tests in automated program repair. Through extensive experiments, we show that (1) flaky tests are quite common in real-world bugs, (2) flaky tests can negatively influence the capability of fault localization tools, and (3) flaky tests can lead to the sharp decrease of repair performance of APR tools. Further investigation shows that the observed degradation in repairability can be caused by (1) the decrease of fault localization performance and (2) the failing tests in the regression test. Such findings call for more in-depth analysis for better applying APR techniques for solving real-world defects. We call on a community effort for exploring this direction, even though we find paying attention to partial fixes may alleviate the problem brought by flaky tests.

**Artefacts:** All data in the study are publicly available at:

http://doi.org/10.5281/zenodo.4139498

**ACKNOWLEDGEMENTS**

This work was supported by the Project 1015-YAH20102, the Open Project Program of the State Key Laboratory of Mathematical Engineering and Advanced Computing, the National Natural Science Foundation of China (Grant No.61672529 and No.61802180), the Natural Science Foundation of Jiangsu Province (Grant No.BK20180421), the National Cryptography Development Fund (Grant No.MMJJ20180105) and the Fundamental Research Funds for the Central Universities (Grant No.NE2018106).
REFERENCES


