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
†Nanjing University of Aeronautics and Astronautics, Nanjing, China, kui.liu@nuaa.edu.cn
‡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 ($\approx$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 repairability 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 relax the artificial assumption that the test suite is free from flaky tests. One direction that we propose is to consider developing strategies where patches that partially-fix bugs are considered worthwhile: a patch may make the program pass some test cases but fail some (which may actually be the flaky ones).**

***Index Terms*—Program Repair, Flaky Tests, Empirical Assessment**

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]).

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 and investigate the proportion of flaky tests within the real-

*Shangwen Wang and Kui Liu are corresponding authors.

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 *async wait*, *concurrency* and *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 life-cycle 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 *flaky method*. Listing 1 gives an example in which a test method *testFindDomainBounds* within the test file *TimeSeriesCollectionTests.java* is silent. We also note that this phenomenon has been questioned before[1] and the developers of Defects4J claim that removing these tests is for ensuring that this framework always returns reliable results.

```
// Defects4J: flaky method
//   public void testFindDomainBounds() {
//     TimeSeriesCollection dataset = new
    TimeSeriesCollection();
//     List visibleSeriesKeys = new java.util.ArrayList();
//     Range r = DatasetUtilities.findDomainBounds(dataset,
    visibleSeriesKeys,
//             true);
//     assertNull(r);
//     ......
```

Listing 1: A Silent Flaky Test in the Bug Chart-1.

We adopt a heuristic approach to collect the flaky tests. After checking out each bug, we use the string *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 commend *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.

[1]https://github.com/rjust/defects4j/issues/355

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 I: Numbers of flaky bugs and flaky tests in Defects4J.

Project	flaky bugs	#bugs	#flaky tests (unique)†	#tests†
Chart	1-26	26	305 (82)	60 820
Closure	1-62,64-92,94-133	131	501 (31)	952 347
Lang	1,3-65	64	864 (32)	119 561
Math	1-102	106	457 (14)	258 005
Mockito	1-38	38	1 237 (74)	44 067
Time	1-20,22-27	26	576 (25)	101 361
Total	387	391*	3 940 (258)	1 536 161

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., ≈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 ≈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.

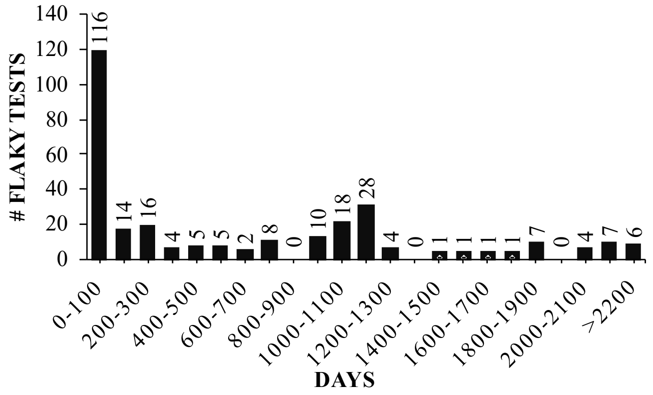


Fig. 1: The distribution of lifespan of flaky tests.

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. Note that in this figure we only show the number of unique flaky tests, which leads to the inconsistency with Table I where a flaky test occurring in diverse bugs is summed for multiple times.

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., ≈ 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

TABLE II: Results of 10 times running of flaky tests under diverse environments.

		jdk1.7 + ubuntu16.04				jdk1.8 + ubuntu16.04				jdk1.7 + ubuntu18.04				jdk1.8 + ubuntu18.04			
Project	**NFT**	**P**	**F**	**R**	**T**	**P**	**F**	**R**	**T**	**P**	**F**	**R**	**T**	**P**	**F**	**R**	**T**
Chart	305	0	305	0	0	0	305	0	0	0	305	0	0	0	305	0	0
Closure	501	418	83	0	0	119	382	0	0	418	83	0	0	119	382	0	0
Lang	864	456	407	1	0	429	434	1	0	455	407	2	0	428	434	2	0
Math	457	279	158	2	18	222	216	1	18	279	158	2	18	220	216	3	18
Mockito	1237	870	339	28	0	550	310	39	338	804	304	129	0	734	478	25	0
Time	576	575	0	1	0	79	497	0	0	573	0	3	0	79	497	0	0
Total	3940	2598	1292	32	18	1399	2144	41	356	2529	1257	136	18	1580	2312	30	18

*NFT denotes number of flaky tests. **P**, **F**, **R**, **T** denote **Pass**, **Fail**, **Random**, and **Timeout**, respectively.

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.

Finding-4 ☞ *Some environment variables diversely influence the results of flaky tests executions. The jdk version is the major factor that we identified in this study (with Defects4J) as impacting the testing results. In comparison, the operating system choice has negligible impact.*

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% (2 765/3 940) 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 (1 497 vs. 1 248).

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 ≈86% (1 005/1 175) 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, **R** tests become **F** 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 fauilt 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 run-time 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

TABLE III: Comparison between running results of flaky tests under jdk-1.7 & Ubuntu-18.04 and jdk-1.8 & Ubuntu-18.04.

Project	All	PP	FF	RR	TT	PF	PR	FP	FR	RP	RF
Chart	305	0	305	0	0	0	0	0	0	0	0
Closure	501	111	75	0	0	307	0	8	0	0	0
Lang	864	426	407	0	0	27	2	0	0	2	0
Math	457	219	158	1	18	58	2	0	0	1	0
Mockito	1237	665	303	1	0	116	23	0	1	69	59
Time	576	76	0	0	0	497	0	0	0	3	0
Total	3940	1497	1248	2	18	1005	27	8	1	75	59

*For a flaky test, **AB** denotes its testing result under jdk-1.7 & Ubuntu-18.04 is **A** while under jdk-1.8 & Ubuntu-18.04 is **B**.

ranked at first place. On the contrary, a statement is considered as not located if the calculated suspicious value is 0.

We consider results obtained without flaky tests as baselines and make comparisons against them using results where flaky tests are included, which are shown in Table IV.

Results reveal a number of findings. We first note that flaky tests can seldom help improve fault localization effectiveness. Specifically, when executed under jdk-1.7, none of the bugs experiences an improved location result while 90 of the bugs get worse fault localization results. With jdk version changing to 1.8, results of three bugs are improved with more bugs (i.e., 141) suffering from retrogress at the same time. Three outliers in jdk-1.8 are *Closure-54*, *Mockito-10*, and *Time-25* where taking flaky tests into consideration boosts the performance of fault localization. For example, in *Mockito-10*, the ranking result is improved from 65th to 20th. In order to understand these special cases, we manually investigate the behind reasons which are found to be diverse across different cases:

In *Mockito-10*, the suspiciousness value of the real buggy statement remains unchanged after considering flaky tests, which means the flaky tests do not cover the buggy point. However, the results of flaky tests are passing, leading to the decrease of suspiciousness value for other statements (cause they are covered by more passing tests). As a result, the ranking of the real buggy statement is increased. The situation is different when it comes to *Closure-54* and *Time-25* where failed flaky tests cover the real buggy statements and thus increase the suspiciousness of them. Hence, their location results are improved.

We also note that considerable numbers of bugs are not affected by flaky tests, i.e., results of these bugs are stable before and after taking flaky tests into consideration (120 under jdk-1.7 and 65 under jdk-1.8). Another interesting finding is that for bugs whose buggy part cannot be located without flaky tests, they are still not locatable after including flaky tests, which demonstrates the uselessness of flaky tests for fault localization from another perspective.

Finding-6 ☞ *Flaky tests hardly ever help improve fault localization performance while often prevent buggy statements from being precisely located, e.g., location results of 90 out of 387 bugs decrease while none of the bugs gets a better result.*

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 localizaiton 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 indicates the flaky tests may have limited influence. **Closure** 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.

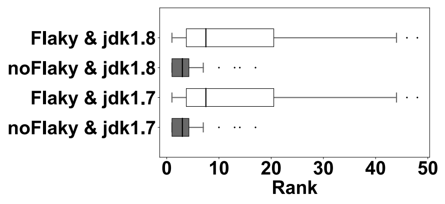
So far we have only investigated the overall situation of fault localization results. We have not yet dissected the changes in the FL results of each locable bug. To achieve so, we define

[2]https://github.com/GZoltar/gzoltar/issues/6

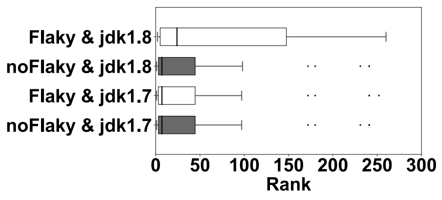
TABLE IV: Fault localization results with flaky tests compared against those obtained without flaky tests.

		jdk1.7 + Ubuntu-18.04					jdk1.8 + Ubuntu-18.04				
Project	**#bugs**	↑	→	↓	**Not located**	**Timeout**	↑	→	↓	**Not located**	**Timeout**
Chart	26	0	5	19	2	0	0	5	19	2	0
Closure	131	0	46	5	80	0	1	16	34	80	0
Lang	64	0	4	22	38	0	0	4	22	38	0
Math	102	0	41	38	5	18	0	39	40	5	18
Mockito	38	0	2	5	31	0	1	1	4	32	0
Time	26	0	22	1	3	0	1	0	22	3	0
Total	387	0	120	90	159	18	3	65	141	160	18

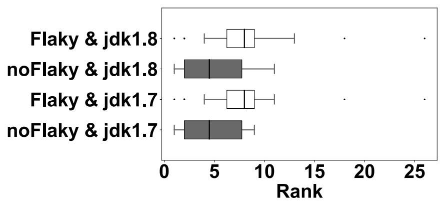
↑ denotes number of bugs whose location results are improved (i.e., rankings of buggy statements are higher), → denotes number of bugs whose location results are unchanged, ↓ denotes number of bugs whose location results are decreased (i.e., rankings of buggy statements are lower), **not located** denotes 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.



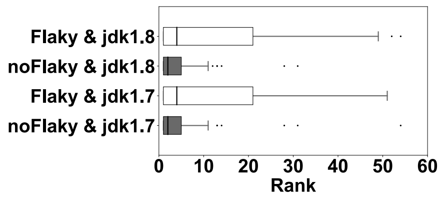
(a) Chart



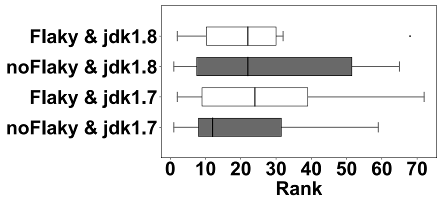
(b) Closure



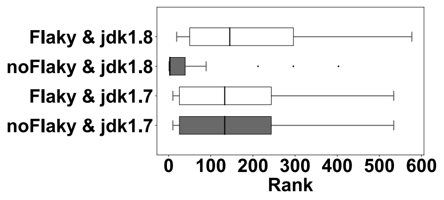
(c) Lang



(d) Math



(e) Mockito



(f) Time

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

a concept *disturbance degree* here to represent the position change in the ranking. For example, if the FL results of a bug is 1st without flaky tests and 10th with flaky tests, then its *disturbance degree* is calculated as 9.

We draw the distribution of disturbance degree in Figure 3. We find that the disturbances are generally limited since the medium values under both conditions are smaller than 20 (i.e., considering flaky tests may drop less than 20 places in rankings for buggy statements under most conditions). Nonetheless, the differences can still be significant under both jdk versions: under jdk-1.8, nearly a quarter of disturbances exceed 40, under jdk-1.7, more specifically, the ranking result of *Chart-1* is changed from 10th to 44th. Considering the fixing attempts made at each suspicious location which utilize different fixing ingredients (i.e., code used to generate patches) and diverse repair actions (i.e., add, delete, update, and move), this would drastically decrease the efficiency of program repair [21].

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.

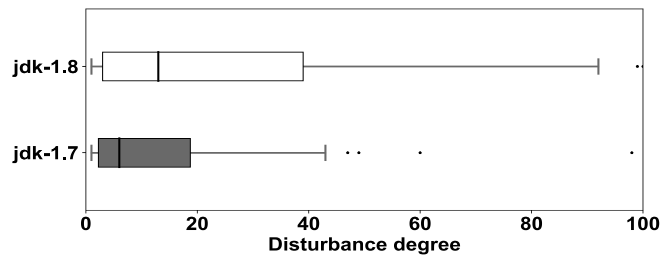


Fig. 3: Distribution of disturbance degree after including flaky tests.

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 loss 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.*

Benchmark overfitting [20] is a phenomenon observed by Durieux *et al.* which refers to the repairability of APR tools is significantly higher for bugs from Defects4J compared to the other benchmarks (i.e., Bears [16], Bugs.jar [64], IntroClassJava [65], and QuixBugs [66]). The authors put forward three potential reasons for this phenomenon which are technical problems in APR tools, bug fix isolation in Defects4J, as well as differences in bug type distribution. We re-calculate the p-values using results with flaky tests, that is to apply Chi-square test on the number of patched bugs for Defects4J compared to the other four benchmarks (the same method as the previous study [20]), and list the new p-values in Table VI. We find that this time, all p-values are larger than the significance level (0.05) except that of *DynaMoth*. We thus accept the null hypothesis for most of the tools that is the number of patched bugs by them is independent of Defects4J. Hence, we propose another potential reason for benchmark overfitting: it happens due to the ignorance of flaky tests in Defects4J benchmark. We can make this hypothesis in that other real-world defect benchmarks (e.g., Bears) do not deal with flaky tests specially.

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 this time. 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.*

TABLE V: Impact on repair performance of APR tools from RepairThemAll framework when flaky tests are considered.

	Chart		Closure		Lang		Math		Mockito		Time		Total		Recall (%)	
Tool	**no flaky**	**flaky**	**no flaky**	**flaky**	**no flaky**	**flaky**	**no flaky**	**flaky**	**no flaky**	**flaky**	**no flaky**	**flaky**	**no flaky**	**flaky**	**no flaky**	**flaky**
Arja	0/9	0/0	1/37	0/2	0/16	0/0	6/22	2/4	0/1	0/0	0/0	0/0	7/85	2/6	21.96	↓20.41
GenProg-A	0/6	0/0	1/22	0/2	0/2	0/0	2/14	0/1	0/0	0/0	0/0	0/0	3/44	0/3	11.37	↓10.59
Kali-A	0/6	0/0	2/49	1/4	0/2	0/0	1/13	0/0	0/1	0/0	0/0	0/0	3/71	1/4	18.35	↓17.31
RSRepair-A	0/8	0/0	1/27	0/2	0/4	0/0	5/22	2/5	0/0	0/0	0/0	0/0	6/61	2/7	15.76	↓13.95
Cardumen	2/5	0/0	0/0	0/0	0/0	0/0	1/12	0/3	0/0	0/0	0/0	0/0	3/17	0/3	4.39	↓3.62
jGenProg	0/7	0/0	1/4	0/2	0/0	0/0	2/20	0/5	0/0	0/0	0/0	0/0	3/31	0/7	8.01	↓6.20
jKali	0/5	0/0	2/11	1/6	0/0	0/0	0/10	0/0	0/0	0/0	0/0	0/0	2/26	1/6	6.72	↓5.17
jMutRepair	1/3	0/0	1/6	0/3	0/0	0/0	2/10	0/1	0/0	0/0	0/0	0/0	4/19	0/4	4.91	↓3.88
Nopol	0/7	0/0	1/69	0/11	1/6	0/0	1/22	0/2	0/0	0/0	0/1	0/0	3/105	0/14	27.13	↓23.51
DynaMoth	0/7	0/0	1/47	0/15	0/2	0/0	1/14	0/1	0/1	0/0	0/1	0/0	2/72	0/16	18.60	↓14.47

x/y means that the APR tool can generate y plausible patches for bugs in this project among which x are correct ones. Data without flaky tests is from a previous study [20].

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

Tool	Bears (251)	Bugs.jar (1,158)	Defects4J (387)	IntroClassJava (297)	QuixBugs (40)	p-value
ARJA	12 (4%)	21 (1%)	6 (1%)	23 (7%)	4 (10%)	0.053
GenProg-A	1 (<1%)	9 (<1%)	3 (<1%)	18 (6%)	4 (10%)	0.138
Kali-A	15 (5%)	24 (2%)	4 (1%)	5 (1%)	2 (5%)	0.060
RSRepair-A	1 (<1%)	6 (<1%)	7 (1%)	22 (7%)	4 (10%)	0.915
Cardumen	13 (5%)	12 (1%)	3 (<1%)	0 (0%)	4 (10%)	0.195
jGenProg	13 (5%)	14 (1%)	7 (1%)	4 (1%)	3 (7%)	0.857
jKali	10 (3%)	8 (<1%)	6 (1%)	5 (1%)	2 (5%)	0.860
jMutRepair	7 (2%)	11 (<1%)	4 (1%)	24 (8%)	3 (7%)	0.067
Nopol	1 (<1%)	72 (6%)	14 (3%)	32 (10%)	1 (2%)	0.058
DynaMoth	0 (0%)	124 (10%)	16 (4%)	6 (2%)	2 (5%)	0.016

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

	no flaky		flaky	
Project	**Fixed**	**Partially fixed**	**Fixed**	**Partially fixed**
Chart	9/14	0/4	1/4	0/13
Closure	7/11	1/5	6/8	1/8
Lang	5/14	0/3	1/4	3/8
Math	19/36	0/4	7/13	7/20
Mockito	1/2	0/0	1/2	0/0
Time	1/3	1/2	1/3	1/2
Total	42/80	2/18	17/34	12/51

***A/B** where A indicates the number of correct patches and B denotes the number of plausible patches.

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 for 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

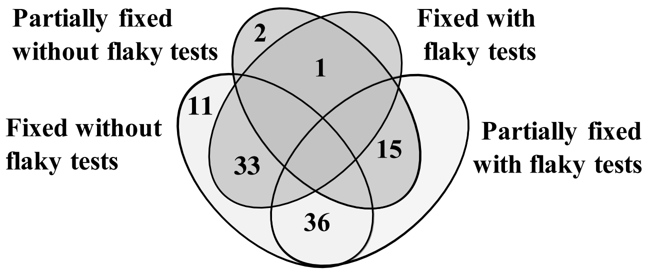

Partially fixed without flaky tests
Fixed with flaky tests
Fixed without flaky tests
Partially fixed with flaky tests
2
1
11
33
15
36

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

bug. Nevertheless, this strategy could only be applied to the situation where reasonable patched can be generated within the time limits (or NPC limits). According to our analysis in Section IV-D2, some bugs cannot be fixed or partially fixed due to the decrease of fault localization results. Hence, how to mitigate the effects of flaky tests on fault localization still needs more exploration.

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.

Threats to internal validity. A major threat to internal validity lies in the selection process of flaky tests: we rely on the manual labels from the developers of Defects4J for identifying flaky tests. Such a method could in theory cause false positives (non-flaky tests labelled as flaky) and false negatives (flaky tests labelled as non-flaky). This, however, is mitigated in that (1) we believe tremendous efforts have been made by the developers of Defects4J to ensure a stable dataset in that a script named *fix_test_suite* is dedicated to identify and remove flaky tests, and (2) the users of Defects4J benchmark also continuously report reproducible results which are further utilized for annotating flaky tests[3]. Furthermore, it

[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 tended 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 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

[1] C. Le Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[2] M. Monperrus, "The living review on automated program repair," in *HAL/archives-ouvertes. fr, Technical Report*, 2018.

[3] K. Liu, L. Li, A. Koyuncu, D. Kim, Z. Liu, J. Klein, and T. F. Bissyandé, "A critical review on the evaluation of automated program repair systems," *Journal of Systems and Software*, 2020.

[4] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "TBar: Revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2019, pp. 31–42.

[5] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2018, pp. 298–309.

[6] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 1–11.

[7] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: combining context-aware neural translation models using ensemble for program repair," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 101–114.

[8] Z. Yu, M. Martinez, B. Danglot, T. Durieux, and M. Monperrus, "Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the nopol repair system," *Empirical Software Engineering*, vol. 24, no. 1, pp. 33–67, 2019.

[9] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, pp. 34–67, 2019.

[10] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 643–653.

[11] M. Fowler, "Eradicating non-determinism in tests," Website, 2020, https://martinfowler.com/articles/nonDeterminism.html.

[12] J. Micco, "The state of continuous integration testing at google," 2017, https://bit.ly/2OohAip.

[13] W. Lam, P. Godefroid, S. Nath, A. Santhiar, and S. Thummalapenta, "Root causing flaky tests in a large-scale industrial setting," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 101–111.

[14] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott, "Sapfix: Automated end-to-end repair at scale," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 2019, pp. 269–278.

[15] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 23rd International Symposium on Software Testing and Analysis*. ACM, 2014, pp. 437–440.

[16] F. Madeiral, S. Urli, M. Maia, and M. Monperrus, "BEARS: an extensible java bug benchmark for automatic program repair studies," in *Proceedings of the 26th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2019, pp. 468–478.

[17] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, M. Monperrus, J. Klein, and Y. Le Traon, "iFixR: Bug report driven program repair," in *Proceedings of the 27the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2019, pp. 314–325.

[18] J. Campos, A. Riboira, A. Perez, and R. Abreu, "GZoltar: an eclipse plug-in for testing and debugging," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2012, pp. 378–381.

[19] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[20] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu, "Empirical review of java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts," in *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2019, pp. 302–313.

[21] K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. Bissyandé, D. Kim, P. Wu, J. Klein, X. Mao, and Y. L. Traon, "On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs," in *Proceedings of the 42nd International Conference on Software Engineering*. ACM, 2020, pp. 615–627.

[22] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *Proceedings of the 39th International Conference on Software Engineering*. ACM, 2017, pp. 609–620.

[23] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The dstar method for effective software fault localization," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290–308, 2013.

[24] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION*. IEEE, 2007, pp. 89–98.

[25] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Transactions on software engineering and methodology (TOSEM)*, vol. 20, no. 3, pp. 1–32, 2011.

[26] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 2014, pp. 153–162.

[27] M. Papadakis and Y. Le Traon, "Metallaxis-fl: mutation-based fault localization," *Software Testing, Verification and Reliability*, vol. 25, no. 5-7, pp. 605–628, 2015.

[28] Z. Zhang, Y. Lei, X. Mao, and P. Li, "Cnn-fl: An effective approach for localizing faults using convolutional neural networks," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 445–455.

[29] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An empirical study of fault localization families and their combinations," *IEEE Transactions on Software Engineering*, 2019.

[30] Y. Lou, A. Ghanbari, X. Li, L. Zhang, H. Zhang, D. Hao, and L. Zhang, "Can automated program repair refine fault localization? a unified debugging approach," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2020, pp. 75–87.

[31] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Transactions on Software Engineering*, vol. 38, no. 1, pp. 54–72, 2012.

[32] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 254–265.

[33] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "AVATAR: fixing semantic bugs with fix patterns of static analysis violations," in *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2019, pp. 456–467.

[34] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proceedings of the 35th International Conference on Software Engineering*. IEEE, 2013.

[35] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. L. Traon, "FixMiner: mining relevant fix patterns for automated program repair," *Empirical Software Engineering*, vol. 25, no. 3, pp. 1980–2024, 2020.

[36] K. Liu, D. Kim, T. F. Bissyandé, S. Yoo, and Y. Le Traon, "Mining fix patterns for findbugs violations," *IEEE Transactions on Software Engineering*, 2018.

[37] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2017.

[38] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: syntax- and semantic-guided repair synthesis via programming by examples," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*. ACM, 2017, pp. 593–604.

[39] Z. Chen, S. J. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair," *IEEE Trans. on Software Engineering*, 2019.

[40] K. Liu, A. Koyuncu, K. Kim, D. Kim, and T. F. Bissyandé, "LSRepair: Live search of fix ingredients for automated program repair," in *Proceedings of the 25th Asia-Pacific Software Engineering Conference ERA Track*. IEEE, 2018, pp. 658–662.

[41] F. Long and M. Rinard, "An analysis of the search spaces for generate and validate patch generation systems," in *Proceedings of the 38th International Conference on Software Engineering*. IEEE, 2016, pp. 702–713.

[42] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proceedings of the 24th International Symposium on Software Testing and Analysis*. ACM, 2015, pp. 24–36.

[43] H. Ye, M. Martinez, and M. Monperrus, "Automated patch assessment for program repair at scale," *CoRR*, vol. abs/1909.13694, 2019.

[44] Q. Xin and S. P. Reiss, "Identifying test-suite-overfitted patches through test case generation," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2017, pp. 226–236.

[45] B. Yang and J. Yang, "Exploring the differences between plausible and correct patches at fine-grained level," in *Proceedings of the 2nd International Workshop on Intelligent Bug Fixing*. IEEE, 2020, pp. 1–8.

[46] S. Wang, M. Wen, B. Lin, H. Wu, Y. Qin, D. Zou, X. Mao, and H. Jin, "Automated patch correctness assessment: How far are we?" in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2020.

[47] H. Tian, K. Liu, A. K. Kaboré, A. Koyuncu, L. Li, J. Klein, and T. F. Bissyandé, "Evaluating representation learning of code changes for predicting patch correctness in program repair," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2020.

[48] X.-B. D. Le, L. Bao, D. Lo, X. Xia, S. Li, and C. Pasareanu, "On reliability of patch correctness assessment," in *Proceedings of the 41st International Conference on Software Engineering*. IEEE, 2019, pp. 524–535.

[49] W. Lam, K. Muşlu, H. Sajnani, and S. Thummalapenta, "A study on the lifecycle of flaky tests," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1471–1482. [Online]. Available: https://doi.org/10.1145/3377811.3381749

[50] M. Eck, F. Palomba, M. Castelluccio, and A. Bacchelli, "Understanding flaky tests: The developer's perspective," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 830–840. [Online]. Available: https://doi.org/10.1145/3338906.3338945

[51] M. T. Rahman and P. C. Rigby, "The impact of failing, flaky, and high failure tests on the number of crash reports associated with firefox builds," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 857–862.

[52] S. Thorve, C. Sreshtha, and N. Meng, "An empirical study of flaky tests in android apps," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 09 2018, pp. 534–538.

[53] F. Palomba and A. Zaidman, "Notice of retraction: Does refactoring of test smells induce fixing flaky tests?" in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 1–12.

[54] A. Gambi, J. Bell, and A. Zeller, "Practical test dependency detection," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, 2018, pp. 1–11.

[55] J. Bell, O. Legunsen, M. Hilton, L. Eloussi, T. Yung, and D. Marinov, "Deflaker: Automatically detecting flaky tests," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, 2018, pp. 433–444.

[56] W. Lam, R. Oei, A. Shi, D. Marinov, and T. Xie, "idflakies: A framework for detecting and partially classifying flaky tests," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, 2019, pp. 312–322.

[57] G. Pinto, B. Miranda, S. Dissanayake, M. d'Amorim, C. Treude, and A. Bertolino, "What is the vocabulary of flaky tests?" in *Proceedings of the 17th International Conference on Mining Software Repositories*, ser. MSR '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 492–502.

[58] M. Zhang, X. Li, L. Zhang, and S. Khurshid, "Boosting spectrum-based fault localization using pagerank," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 261–272.

[59] K. Liu, D. Kim, A. Koyuncu, L. Li, T. F. Bissyandé, and Y. Le Traon, "A closer look at real-world patches," in *Proceedings of the 34th International Conference on Software Maintenance and Evolution*. IEEE, 2018, pp. 275–286.

[60] Y. Yuan and W. Banzhaf, "Arja: Automated repair of java programs via multi-objective genetic programming," *IEEE Transactions on Software Engineering*, vol. 46, no. 10, pp. 1040–1067, 2020.

[61] M. Martinez and M. Monperrus, "ASTOR: a program repair library for java (demo)," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 441–444.

[62] T. Durieux and M. Monperrus, "DynaMoth: dynamic code synthesis for automatic program repair," in *Proceedings of the 11th International Workshop in Automation of Software Test*. ACM, 2016, pp. 85–91.

[63] T. Durieux, B. Cornu, L. Seinturier, and M. Monperrus, "Dynamic patch generation for null pointer exceptions using metaprogramming," in *Proceedings of the 24th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2017, pp. 349–358.

[64] R. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. Prasad, "Bugs.jar: A large-scale, diverse dataset of real-world java bugs," in *Proceedings of the 15th IEEE/ACM International Conference on Mining Software Repositories*. ACM, 2018, pp. 10–13.

[65] T. Durieux and M. Monperrus, "Introclassjava: A benchmark of 297 small and buggy java programs," KTH Royal Institute of Technology, Tech. Rep., 2016.

[66] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "QuixBugs: a multi-lingual program repair benchmark set based on the quixey challenge," in *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*. ACM, 2017, pp. 55–56.

[67] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. L. Traon, "You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems," in *Proceedings of the 12th IEEE International Conference on Software Testing, Verification and Validation*. IEEE, 2019, pp. 102–113.

[68] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering*. IEEE, 2017, pp. 416–426.

[69] X. Liu and H. Zhong, "Mining stackoverflow for program repair," in *Proceedings of the 25th IEEE International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2018, pp. 118–129.

[70] M. Soto and C. Le Goues, "Using a probabilistic model to predict bug fixes," in *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering*. IEEE, 2018, pp. 221–231.

[71] S. Wang, M. Wen, L. Chen, X. Yi, and X. Mao, "How different is it between machine-generated and developer-provided patches? an empirical study on the correct patches generated by automated program repair techniques," in *Proceedings of the 13th International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2019, pp. 1–12.

[72] S. Wang, M. Wen, X. Mao, and D. Yang, "Attention please: Consider mockito when evaluating newly proposed automated program repair techniques," in *Proceedings of the 23rd Evaluation and Assessment on Software Engineering*. ACM, 2019, pp. 260–266.

[73] B. Lin, S. Wang, M. Wen, Z. Zhang, H. Wu, Y. Qin, and X. Mao, "Understanding the non-repairability factors of automated program repair techniques," in *Proceedings of the 27th Asia-Pacific Software Engineering Conference*, 2020.