Simulating the Effect of Test Flakiness on Fault Localization Effectiveness

Béla Vancsics
Department of Software Engineering
University of Szeged, Hungary
vancsics@inf.u-szeged.hu

Tamás Gergely
Department of Software Engineering
University of Szeged, Hungary
gertom@inf.u-szeged.hu

Árpád Beszédes
Department of Software Engineering
University of Szeged, Hungary
beszedes@inf.u-szeged.hu

***Abstract*—Test flakiness (non-deterministic behavior of test cases) is an increasingly serious concern in industrial practice. However, there are relatively little research results available that systematically address the analysis and mitigation of this phenomena. The dominant approach to handle flaky tests is still detecting and removing them from automated test executions. However, some reports showed that the amount of flaky test is in many cases so high that we should rather start working on approaches that operate in the presence of flaky tests. In this work, we investigate how test flakiness affects the effectiveness of Spectrum Based Fault Localization (SBFL), a popular class of software Fault Localization (FL), which heavily relies on test case execution outcomes. We performed a simulation based experiment to find out what is the relationship between the level of test flakiness and fault localization effectiveness. Our results could help the users of automated FL methods to understand the implications of flaky tests in this area and to design novel FL algorithms that take into account test flakiness.**

***Index Terms*—Test flakiness, flaky tests, fault localization, Spectrum-Based Fault Localization, testing and debugging.**

I. Introduction

Fault localization (FL) is a necessary step before any automatic or manual program repair action. The successfulness of program repair is hence dependent on the efficiency of fault localization [1], [2]. There are a lot of FL approaches that help to automatically detect the location of bugs, and one of the largest families is Spectrum-Based Fault Localization (SBFL) [3], [4], [5], [6], our focus in this article.

Current literature on program repair and fault localization typically does not consider the possible flakiness of testing environments; the performance of these approaches is predominantly investigated using bug benchmarks that rely on deterministic test case behavior. Studies about the detection, analysis and treatment of flaky tests reveal that there is a non-negligible amount of tests with flakiness, which negatively impacts current industrial practices [7], [8], [9]. However, it is not known at this time how does flakiness impact fault localization and how these applications could be fit for potentially significant levels of flakiness.

In this paper we present an empirical study to investigate the effects of test flakiness on existing SBFL fault localization techniques (Tarantula [10], Ochiai [11] and DStar [12]), which, we believe, enables a better understanding of how much does flakiness affect the performance of traditional FL algorithms. We define the flakiness-ratio metric to express the flakiness of a test, and conduct a simulation-based experiment using real projects with real faults. The goal is to measure how much fault localization scores are affected by different levels of flakiness of individual test cases. We investigate whether the different localization scores and the ranks of the buggy methods based on these scores are changed notably if a single test case becomes flaky.

Results show that the different investigated algorithms are affected by flakiness at different levels. In general, Tarantula is more sensitive to flakiness than Ochiai and DStar. However, results also indicate that this effect highly depends on the characteristics of the actual project, and the related test cases as well. Our empirical study also addresses the question if different properties of the tests that become flaky or the bugs themselves make any difference in this regard.

In Section II, we describe the goal and motivation of this study. Section III describes how the experiment was prepared and implemented, and in Section IV the results are presented along with a discussion of some possible reasons of the observed results (Section V), and the analysis of the threats to validity (Section VI). In Section VII we elaborate on the related work, and the conclusion is given in Section VIII.

II. Goal and Research Questions

A test case is called *flaky* if its outcome is non-deterministic, i. e. sometimes it passes and sometimes it fails depending on unknown circumstances. Flakiness is a known phenomenon in software testing [13], [8], and requires attention. The effect of flakiness has also been investigated for some testing applications [14], [15].

We conducted an experiment in which we investigated how the flakiness property of tests affected the performance of FL algorithms and whether certain patterns could have been identified using the characteristics of bugs and defective methods. In this experiment we simulated the flakiness of individual test cases in 28 versions of the Mockito program (part of the Defect4J bug database [16]) with known bugs. In the simulation, we assumed 100 executions of each test case and assumed that some of these executions changed its original result. To express the flakiness of a test case τ, we defined the *flakiness-ratio* (*FR*) as shown in Eq. 1.

$$FR(\tau) = 2\frac{\min(\tau_P, \tau_F)}{\tau_P + \tau_F} \quad (1)$$

In Eq. 1, τ_P is the number of passed executions of test case τ and τ_F is the number of failed executions of test case τ. For example, if the $test\_foo$ test case passed 70 times and failed 30 times of the 100 executions, then $FR(test\_foo)$ = $2\frac{\min(70,30)}{100}$ = 0.6. If the test is not flaky (i. e. deterministic), its flakiness-ratio is 0; if the test passes and fails equally, its flakiness-ratio is 1. The ratio is symmetric in the sense that the original result of the test itself does not affect the value of this ratio (neither if it was a pass or a fail).

By conducting the experiment, we wanted to answer the following research questions regarding the effect of flakiness on the FL algorithms:

RQ-1: What is the effect of a single flaky test case on the FL algorithms? At what flakiness-ratio the rank of the faulty method(s) change notably? Is there a difference between the different algorithms in their sensitivity to flakiness?

RQ-2: What other features/properties affect the performance of the algorithms? Does the effect change if there is only one or if there are more than one normally failing test cases? Does the number of faulty methods (one or more) makes notable difference? How different FL algorithms work if only passed or only failed tests are flaky?

III. Study Design and Data Preparation

In this section we describe how the experiment was conducted.

A. Measurement Scenarios

We used the Mockito project from Defecst4J [16] as the System Under Tests for the measurements. The program had different faulty versions with 38 bugs known in the code (functions) and the set of deterministic (non-flaky) test cases. We executed fault localization for the bugs with simulated flakiness.

First, for all bugs, we executed the test cases assigned to that program version. During this execution method coverage was also computed (it is required for FL). We used a bytecode instrumentation tool which is based on *Javassist* to collect the necessary information about the code coverage and test results. The minimum requirement for Javassist is Java 5, so we had to skip those versions where the instrumentation were unsuccessful. A total of 28 bugs met these requirements, these cover broad spectrum of different cases / properties (low vs. high bug-ranks, one vs. more failed tests, one vs. more faulty methods). After we had the results (i. e. what test cases passed and failed) and the method coverage (what methods had been called at least once during a test case), we simulated flakiness in the following way.

First, we assumed 100 executions of each test case. The original, deterministic, non-flaky measurement is equivalent with the situation that all 100 runs of a single test case produced the same result. Then, the tests were re-run simulating the situation that a certain number of runs of that test case changed its outcome. We gradually incremented the number of runs that changed their outcomes with an increment of 5 runs; i. e. we first assumed that 95 runs produced the same result and 5 produced the opposite, then we assumed 90-10 ratio, then 85-15, etc., until the 50-50 ratio. The results have been modified artificially, that is, the outcome of a test was changed from passed to failed (or failed to passed), one test case at a time, but the code has not been altered. The flakiness-ratio metric value (FR) of the test case is FR = 0.0 for the original run, FR = 0.1 for the 95-5 ratio, FR = 0.2 for the 90-10 situation, and at the end FR = 1.0 for the 50-50 ratio.

Finally, we performed fault localization for the bugs with these simulated results using the algorithm below.

B. Fault Localization

There are several kinds of FL approaches based on the information they use. In this work, we have chosen Spectrum-Based Fault Localization (SBFL) methods, which capture program execution data. They monitor the behavior of the program and compare this execution data to the test results to locate the erroneous code element(s). The FL formulas we used in this experiment use code coverage to capture program execution. In general, code coverage shows what code elements are executed during the run of a test case.

In our case, we computed method level code coverage, which tells us what methods have been called during the execution of a test case. To capture this coverage information, we produced the *coverage matrix* of the faulty program version. The rows of the matrix represent the tests (T), and the columns are assigned with the functions (F). The coverage matrix has a value of 1 in a given $c_{\tau,\phi}$ position if code element ϕ is covered by test case τ, otherwise this value is 0.

The used FL formulas also need the *result vector*. The results vector contains the pass-fail results of the tests, where 0 means pass and 1 means fail.

Using the coverage matrix and results vector, four basic numbers can be determined for each $\phi \in F$ functions:

a) ϕ_{ep}: number of passing tests covered by ϕ,
b) ϕ_{ef}: number of failing tests covered by ϕ,
c) ϕ_{np}: number of passing tests not covered by ϕ and
d) ϕ_{nf}: number of failing tests not covered by ϕ.

These numbers are then used by several SBFL formulas to compute FL suspiciousness scores for the functions [4], [17], [18]. These scores are then used to order the functions and compute their final suspiciousness rank. The three popular formulas we utilized in this experiment are DStar [19], Ochiai [11], and Tarantula [10]:

$$\text{DStar}: \quad \frac{\phi_{ef}^2}{\phi_{ep} + \phi_{nf}},$$

$$\text{Ochiai}: \quad \frac{\phi_{ef}}{\sqrt{(\phi_{ef} + \phi_{nf}) \cdot (\phi_{ef} + \phi_{ep})}},$$

$$\text{Tarantula}: \quad \frac{\frac{\phi_{ef}}{\phi_{ef}+\phi_{nf}}}{\frac{\phi_{ef}}{\phi_{ef}+\phi_{nf}} + \frac{\phi_{ep}}{\phi_{ep}+\phi_{np}}}$$

To overcome the frequent situation when more functions get the same suspiciousness score, there are three possible decisions. All methods with the same suspiciousness score might have (1) the average rank, (2) the minimum rank, or (3) the maximum rank of the given methods. In this work, we use average ranks.

C. Modified Metrics

The above mentioned metrics are not prepared for flaky results. In our case, the result of a test case is probabilistic, representing not just a single run, but a series of test executions. Besides 0 (passed) and 1 (failed), the result can be any value between them, meaning that the test is more or less flaky. Thus, it was necessary to redefine/extend the above mentioned four numbers and formulas to non-integers.

a)

$$\phi_{ep}' := \sum_{\tau \epsilon T} \Theta(\tau) \text{ and } \tau \text{ covered by } \phi$$

b)

$$\phi_{ef}' := \sum_{\tau \epsilon T} \Gamma(\tau) \text{ and } \tau \text{ covered by } \phi$$

c)

$$\phi_{np}' := \sum_{\tau \epsilon T} \Theta(\tau) \text{ and } \tau \text{ not covered by } \phi$$

d)

$$\phi_{nf}' := \sum_{\tau \epsilon T} \Gamma(\tau) \text{ and } \tau \text{ not covered by } \phi$$

$$\Theta(\tau) = \frac{\tau_P}{\tau_P + \tau_F} = \begin{cases} \frac{FR(\tau)}{2} & \text{if } \tau_P < \tau_F \\ 1 - \frac{FR(\tau)}{2} & \text{if } \tau_P \geq \tau_F \end{cases}$$

$$\Gamma(\tau) = \frac{\tau_F}{\tau_P + \tau_F} = \begin{cases} 1 - \frac{FR(\tau)}{2} & \text{if } \tau_P < \tau_F \\ \frac{FR(\tau)}{2} & \text{if } \tau_P \geq \tau_F \end{cases}$$

The "number of passed" and "number of failed" numbers are no longer counts, but sums of the probabilistic test results. $\Theta(\tau)$ and $\Gamma(\tau)$ simply give the probability that test case τ passes and fails, respectively. These definitions are backward compatible with the original ones, but can handle flakiness.

To determine the rank of the test cases according to their scores, we use the above mentioned average rank strategy.

IV. Results

A. Overall Effect on FL

In the experiment, for each bug, we changed the results of each of its tests individually. Given a bug, for each test case τ we set the flakiness-ratio of τ between 0 and 1 with a 0.1 increment leaving the flakiness of other test cases to be 0. Using these values, we computed FL scores and rankings for all $(\tau, FR(\tau))$ configurations of the bug. Note that all $(\tau, 0)$ configurations are the same and gives the original, non-flaky results. Then, for each configurations, we examined the difference between the original and modified rank of the faulty method(s) using the three formulas DStar, Ochiai, and Tarantula.

Figure 1 shows how the performance of the three formulas change as a function of *FR*. For each bug and *FR* level, the figures show the average relative difference for all configurations of that *FR* level (i. e., for a bug β, for an *FR* level x, it shows average relative rank differences for all τ test cases of β).

Comparing the results, several observations can be made:

1) Tarantula is more sensitive to *FR* than Ochiai and DStar, so even with a lower flakiness value, there may be a more notable difference in rank,
2) Ochiai and DStar produced very similar results,
3) there were bugs where Ochiai and DStar could improve the original rank (diff(%) $\leq$ 100),
4) even with a relatively high *FR* value, the average rank-diffs were moderate in most cases,
5) for different bugs the results are very different, proposing that specific properties of the tests and/or bugs have high influence on the FL results.

Flakiness-sensitivity is well illustrated by the histogram (Figure 2), which shows how many $(\tau, 1.0)$ configurations (i. e. when the result of a single test τ is equally random) produced different changes in the rankings. It can be seen that in some cases (7.2% and 3.6% of the configurations) Ochiai and DStar improved (i. e. the flakiness of the test helped fault localizationation), and in most cases (82.2% of the configurations) the new rank was in the ±25% of the original. It is also obvious that Tarantula responded differently to flakiness: in about 18% of the configurations the ranks were at least doubled.

> **RQ-1:** In the case of flaky tests, all FL formulas give different scores and ranks to the faulty method(s). In general, Tarantula is more sensitive to flakiness than Ochiai and DStar, and starts to change notably at lower flakiness levels. There is no single flakiness-ratio where the algorithms start to produce notably worse results and all three formulas have bugs that behave differently than the other bugs.

B. Effect of Test Properties

To answer **RQ-2**, we filtered the FL results by 3 properties: the number of originally failed test cases, the number of faulty methods, and whether the flaky test originally failed.

First, we checked the difference between bugs having only one failing test case and bugs having more failing test cases. There were 15 bugs with one, and 13 bugs with more failing test cases. The results of flakiness measurements are shown on Figure 3 for a single failing test case and on Figure 4 for more failing test cases. As the results show, in the case of a single failing test case flakiness have a moderate effect on the ranks (less then 20% away from the original in most cases), while in the case of more failing test cases there can be a notable difference (more than 50% worse in more cases).

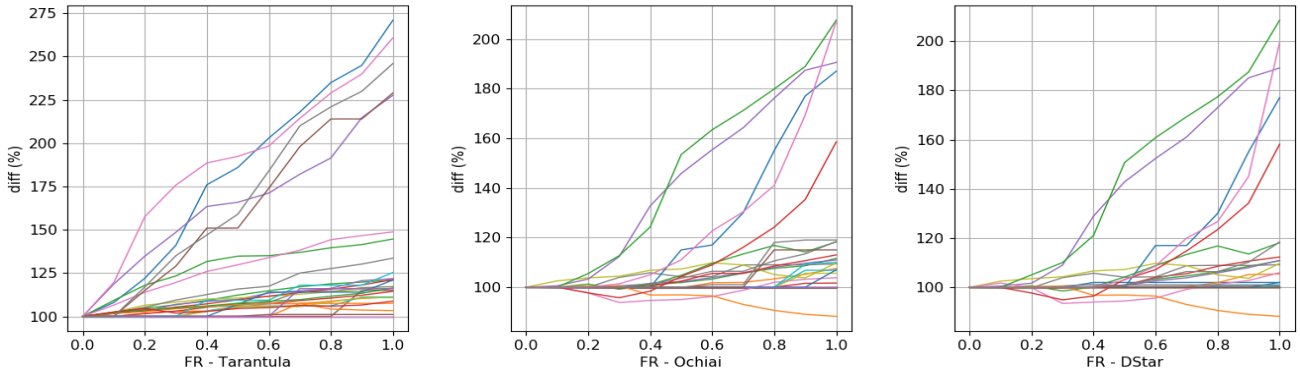

Fig. 1: Tarantula, Ochiai and DStar results (one line represents the results for one bug)

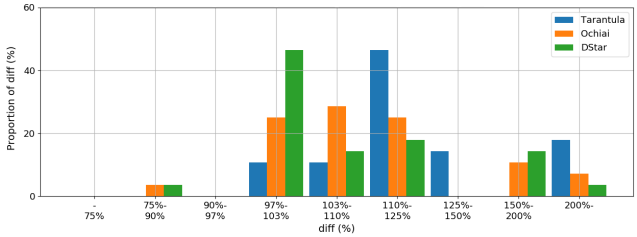

Fig. 2: Tarantula, Ochiai and DStar diff - FR:1.0

The next feature we examined was the number of faulty methods associated with the same bug. Again, we defined two categories: bugs with a single associated faulty method and bugs with multiple faulty methods. There were 18 bugs with a single faulty method and in 10 cases the bug occurred in more than one methods. The FL results of these groups are shown in Figure 5 (single method) and Figure 6 (multiple methods). The observed result is not really surprising: for bugs with multiple faulty methods the algorithms are less sensitive to flakiness. The average new ranks in all but one versions are within a 25% range of the original rank. On the other hand, if there is only one faulty method associated with the bug, the difference between the original and average flaky rank is much larger even for relatively small (e. g. 0.4) *FR* values. We think that this can be caused by the larger set of faulty methods, which can "smooth" the noise introduced by a single flaky test.

We also examined how FL rankings change when an originally passing or originally failing test case becomes flaky. The results of these experiments are shown in Figure 7 (only failed tests are flaky) and Figure 8 (only passed tests are flaky). The results are interesting. For both failed and passed flakies, Tarantula behaves differently than Ochiai and DStar, which are similar.

If the failed test cases become flaky, Tarantula formula improves the average ranks a bit (less than 20%) at higher flakiness in most of the cases, however, for two bugs it produces notably worse results (around 50% points and 110% points worse than the original rank). In the flaky failed cases Ochiai and DStar formulas produce average ranks which are ±25% around the original averages, except for a single bug for which they result in more than 150% worse ranks in average.

If the passed test cases become flaky, the situation is different. In this case Tarantula produces much (more than 100%) worse ranks for five bugs as the flakiness-ratio gets higher, and there is no bug for which higher flakiness would result in better ranks. The relation between the flakiness-ratio and the increase in rank is mostly linear. Ochiai and DStar produces similar results to Tarantula, but the average ranks are usually less than 100% worse than the original averages, and for some bugs, improvements can be observed.

Another possible aspect of the categorization is the ϕ_{ep} value of the faulty methods, i. e. how many passing test case executes the faulty methods. Studies [20], [11] have shown that FL algorithms are less effective in finding faulty methods with high ϕ_{ep} values, i. e. when high number of test cases execute the faulty methods without failure. Based on the original (non-flaky) ϕ_{ep} values, we created two categories of the bugs: we classified 9 bugs (program versions) as *high*-ϕ_{ep}, where $\phi_{ep} \geq 100$, and 19 bugs as low-ϕ_{ep}, where $\phi_{ep} < 100$.

By examining the results on Figure 9 (low-ϕ_{ep} rank change distribution at $FR = 1$ level) and Figure 10 (high-ϕ_{ep} rank change distribution at $FR = 1$ level), we found that there was no notable difference between the results of the two groups using the Tarantula algorithm. However, the results with Ochiai and DStar have shown observable differences. When the ϕ_{ep} value is low, there is ±25% difference between the original and the flaky ranks in the vast majority of cases (94.8%). However, for versions with high ϕ_{ep} value, i. e. in which more than 100 passing test cases exercise the faulty methods, this proportion dropped to 55.6%. In 22.2% of the bugs the average difference between the original and flaky ranks were more than 100%, but at the same time, in about 10% of the cases, Ochiai and DStar produced improvements in the ranks.

RQ 2: All algorithms are more sensitive to flakiness if more than one test case fails for the bug, if there is only one faulty method in the system, or if the failing test cases become flaky. Tarantula seems to be less

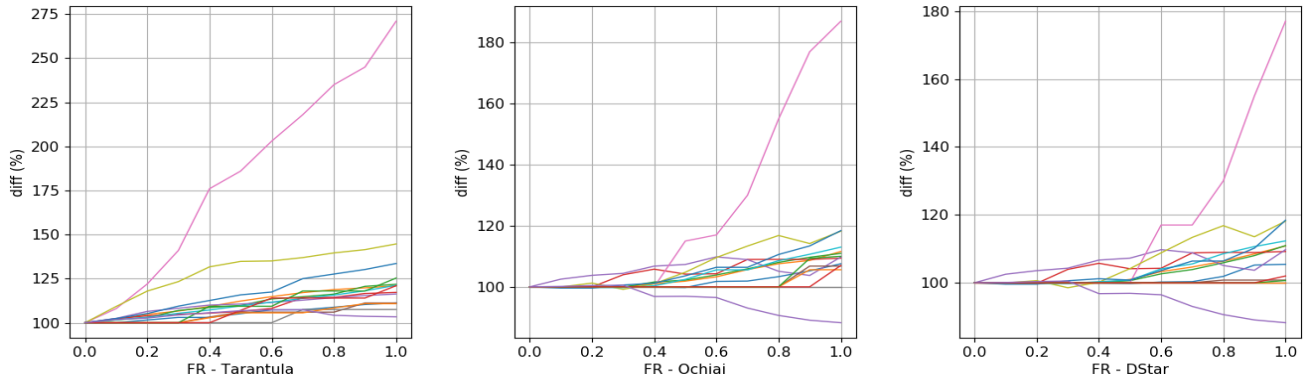

Fig. 3: Results – one failed test (one line represents the results for one bug)

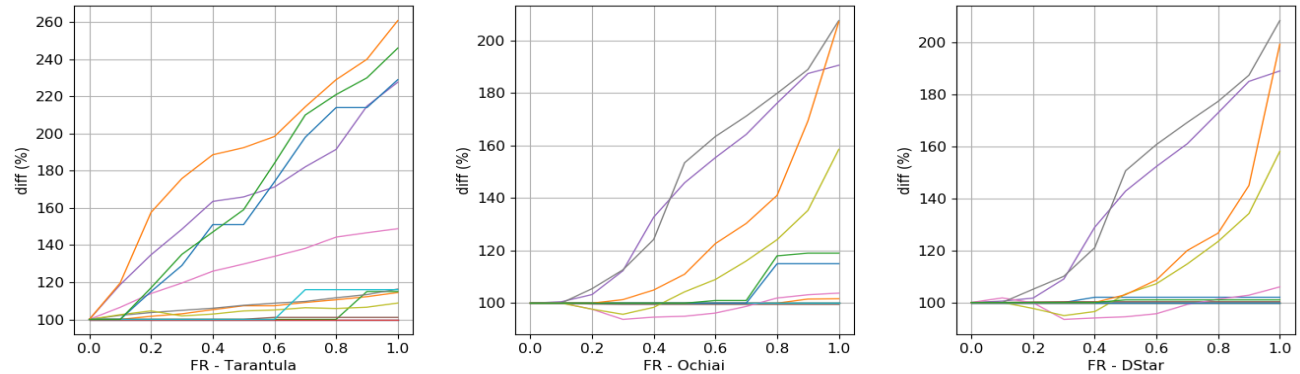

Fig. 4: Results – more failed tests (one line represents the results for one bug)

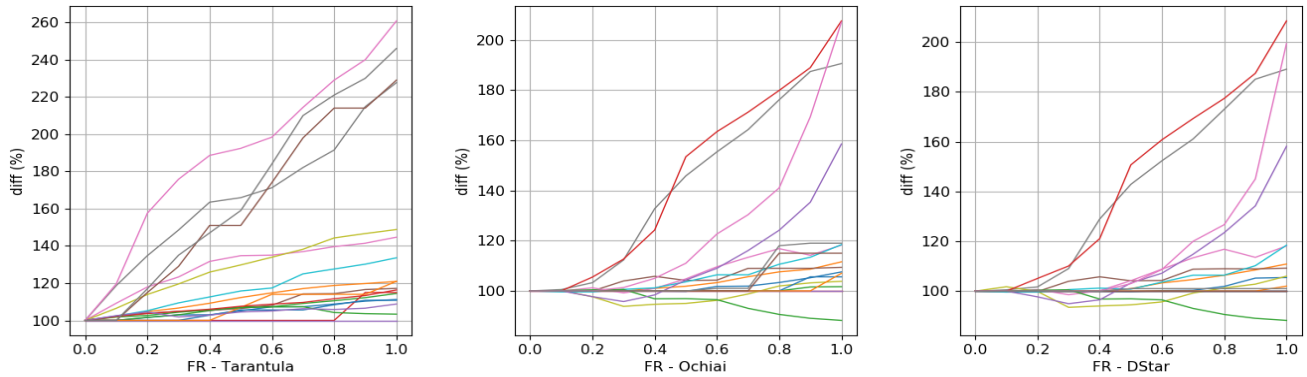

Fig. 5: Results – one faulty method (one line represents the results for one bug)

sensitive to whether the number of covering passing test cases are low or high, while Ochiai and DStar change the ranks differently in these cases.

V. Discussion

It was not among the goals of this article to examine the results in detail to find the root causes of our findings. However, we have some preliminary ideas, which are topics for future work.

First, Ochiai and DStar improve the ranks in some cases, which can be attributed to the method level resolution, where a test can pass even if it executes the faulty method (known as coincidental correctness). Flakiness can reduce this effect thus helping the formulas.

Next, Tarantula is more sensitive to flakiness, which might be the result of the use of ϕ_{np} in the formula. ϕ_{np} is the amount of passed tests not executing the faulty method. If the faulty method is executed by only a small portion of the tests, flakiness can have a more significant effect on the final score of the faulty method.

Finally, the figures show that even with high *FR* value the ranks may not change significantly. We can think of more explanations for this phenomenon. As changes in the FL score

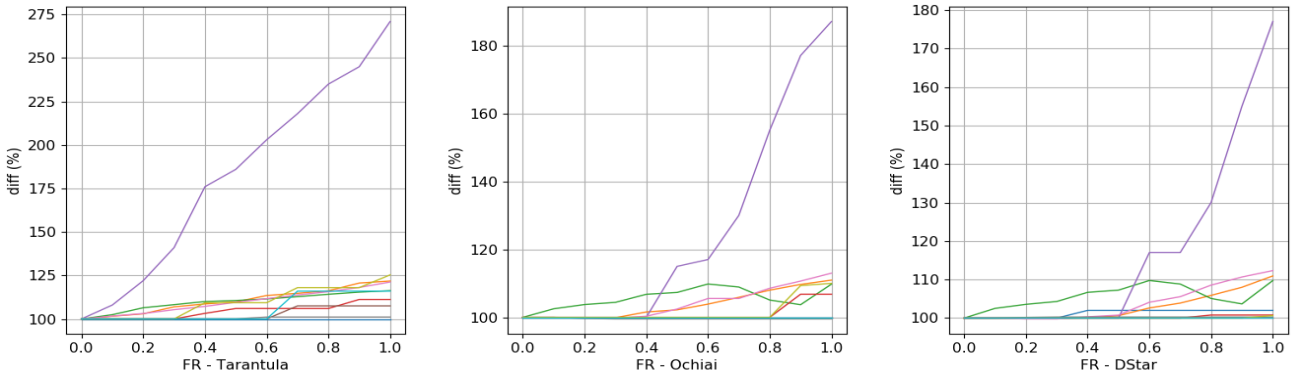

Fig. 6: Results – more faulty methods (one line represents the results for one bug)

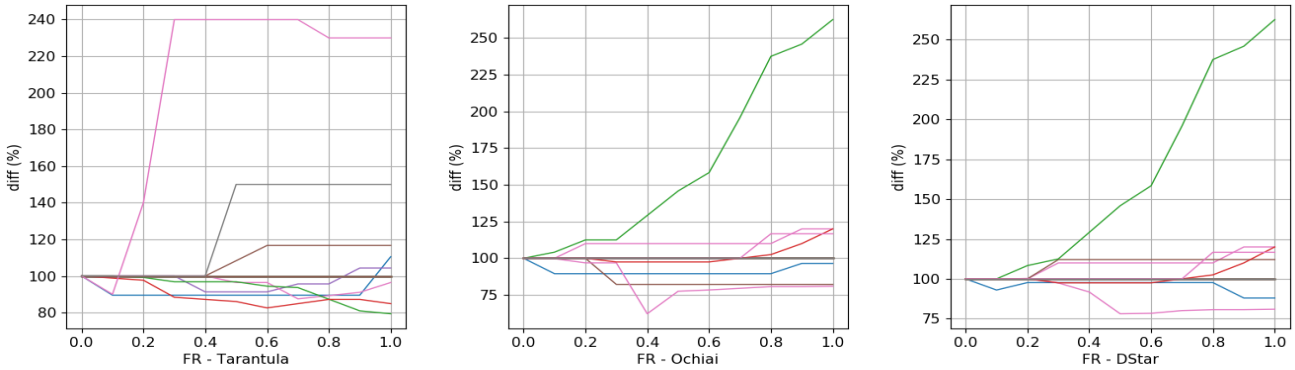

Fig. 7: Tarantula, Ochiai and DStar – failed tests (one line represents the results for one bug)

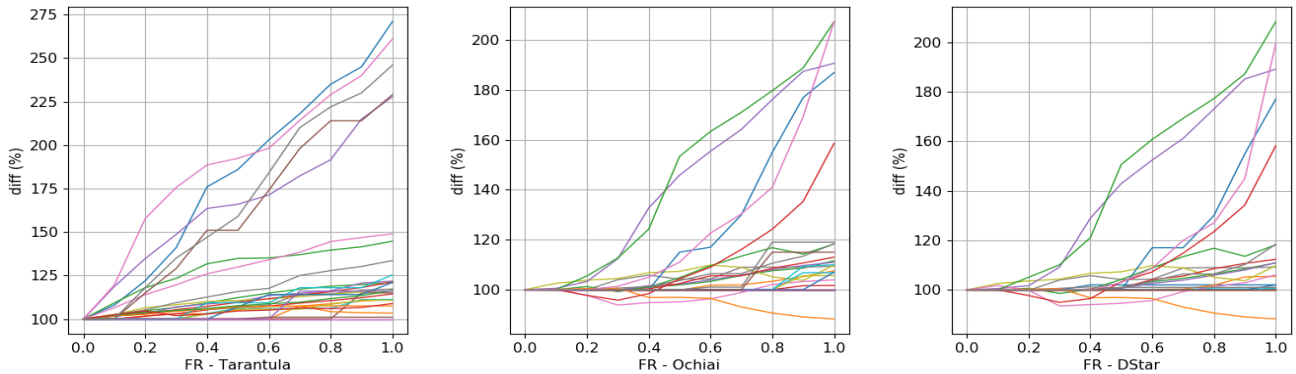

Fig. 8: Tarantula, Ochiai and DStar – passed tests (one line represents the results for one bug)

do not necessarily modify the rank, this might hide the effect of flakiness in individual cases. Furthermore, we present only averages which can also hide individual changes.

VI. Threats to Validity

It is a threat to validity that we only simulated the results, but did not change the associated coverage. The fact that we are working with method level coverage mitigates this threat, however, it is still possible that a real flaky test would produce different coverage for its pass and fail runs and it would modify the computed rankings.

Another threat is that we did not examine all bug cases manually. It might happen that some not examined core features of the subject project versions have an influence on the examined properties. For example, in case of bugs with more failing test cases the ranks can drop severly while in case of bugs with a single test case the rank change is moderate. However, the root cause of this phenomenon can be something else whose consequence was the number of failing test cases in the given versions, and we only detected the co-occurrence of these consequences.

In practice, not only one test can be flaky in one program version. In these cases, flakiness can have a much more

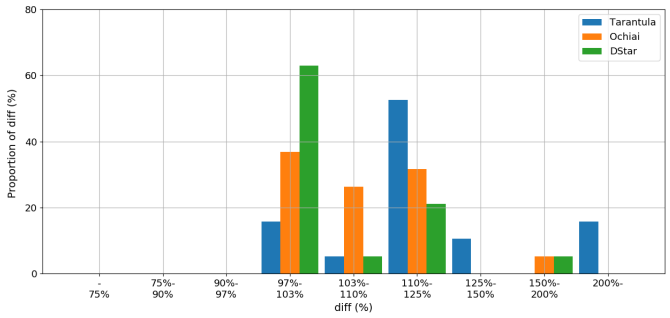

Fig. 9: Tarantula, Ochiai and DStar – low ϕ_{ep}

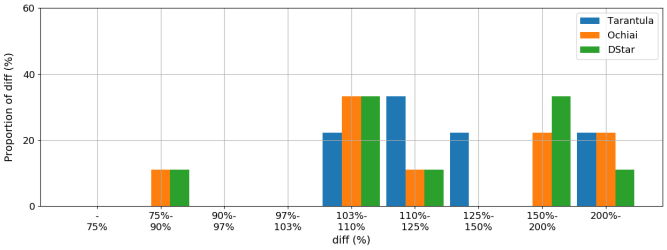

Fig. 10: Tarantula, Ochiai and DStar – high ϕ_{ep}

complex effect on FL efficiency, but due to the limitations of the study, we did not investigate this aspect.

VII. Related Work

A. Fault Localization

Fault localization is a well-researched area with extensive literature [21], [5], [22]. There are a lot of algorithms, all of which are intended to determine the exact location of the bugs. One of the largest fault localization algorithms family is the Spectrum-Based FL (SBFL or Coverage Based Statistical Fault Localization – CBSFL). The essence of these methods are that the behavior of the program and thus the possible location of the bug can be deduced from the collected execution data.

One of the most popular SPFL methods is Tarantula [10], [3]. It prioritizes the methods using the coverage information and the test results. The algorithm orders the methods based on the ratio of the covering failed tests and the not covering failed tests, as well as the covering and not covering passed tests.

Abreu et al. used the Ochiai method in their studies ([23], [11], [24]). This formula was adapted from molecular biology. It was shown that Ochiai produces better results than Tarantula using the Siemens and the SIR bug dataset.

Wong et al. [19] presented the DStar technique, which was evaluated in 24 programs and compared the results of the algorithm with 38 different techniques. Single-fault and multi-fault programs are used for assessment. Empirical evaluation has shown that DStar is better than all other methods.

There are many comparative studies [24], [25], [12] that compare the results of different algorithms. These studies came to the following conclusions: (1) there is difference in efficiency of the algorithms for injected and real bugs, (2) Ochiai performed better than Tarantula, (3) DStar was better than Ochiai.

B. Flaky Tests

Flaky tests have an extensive literature and there are several studies about the causes, effects, and identification of flaky tests [26], [7], [15], [14], [27], [8].

Lam et al. [13] described their experience with flaky tests by conducting a study on them. They identified flaky tests, investigated their root causes and described them to help the developers to avoid and/or fix flakiness. Five non-open source (anonymized) real-world projects were analyzed, they collected all relevant code to log various runtime properties, examined the differences between passing and failing runs and publicized them.

In their empirical study, Luo et al. [8] analyzed and clasifed the most common root causes, described behavior, and presented flaky tests fix-strategies. They defined 12 causes and implications based on 51 examined open-source projects using version-control commits and bug reports. The authors manually grouped the semi-automatically identified cases/causes into existing groups in the literature and, if necessary, created new ones, then analyzed the manifestations and possible improvement options.

A general method for identifying flaky tests is to run them again, but this can often take a long time. Bell et al. [7] presented a new technique (DeFlaker) wich can detect them without rerunning. The method is based on the relationship between coverage change and test result change: if a test passed becomes failed and no coverage change happened, this indicates that it is a flaky test. To validate the method, the authors carried out experiments using 10 real projects and their history, which resulted in only 1.5% false positive cases and 95.5% recall.

FlakiMe [15] is a similar approach/methodology to the one we used. The effect of flakiness on mutation testing and program repair was investigated by Cordy et al. using injected test results modification. The authors also used Defects4J as a benchmark. They concluded that flaky tests reduce the effectiveness of (deterministic) repair techniques by 5% to 100%. One of their related findings was that the non-deterministic tests decreased effectiveness more when the generated patches were covered by more tests. This is consistent with one of our result, that is, the more tests cover the faulty method, the more sensitive the fault localization method is to flaky tests.

VIII. Conclusion

In this work we have examined how the flakiness of test cases affect the performance of fault localization algorithms. We first defined the flakiness-ratio metric to express the flakiness of a test case, and modified the computation of the base $\phi_{ep}, \phi_{ef}, \phi_{np}, \phi_{nf}$ numbers required by the examined fault localization formulas in a way that they remain compatible for non-flaky tests but can handle flakiness.

We conducted an experiment in which we simulated flaky tests for a program with 28 faulty versions, and checked how

different flakiness-ratio values affect the three FL algorithms. We found that one algorithm (Tarantula) is more sensitive to high flakiness of individual test cases than the other two (Ochiai and DStar), i. e. it produces worse ranks relative to the original ones than the other algorithms at the same flakiness-ratio.

We also examined how some features and properties of the program versions and the tests affect the behavior of these algorithms in case of flakiness. We found that the number of faulty methods, the number of failing test cases, the number of passing tests that cover the faulty methods, and the original outcome of the flaky test can all be used for classification where (at least some of) the algorithms produce distinct behavior in case of flaky tests.

It can be a future work to examine the absolute ranks produced by these algorithms in flaky situations, as it might happen that even if the flaky rank is the double of the original one for an algorithm at a certain flakiness-ratio, it is still better than the flaky rank produced by an other algorithm, which is much closer to its original rank. Another possible future work is to manually examine the flaky results of individual bugs in details, as the same algorithm can produce divergent results for different bugs. It is also an interesting question how more flaky tests at the same time influence the performance of the FL algorithms. Finally, more attributes of the tests and test executions could be examined whether they have some effect on the flakiness.

Acknowledgments

This work was partially supported by the EU-funded Hungarian national grant GINOP-2.3.2-15-2016-00037 titled "Internet of Living Things" and by grant TUDFO/47138-1/2019-ITM of the Ministry for Innovation and Technology, Hungary.

References

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

[2] D. Yang, Y. Qi, and X. Mao, "An empirical study on the usage of fault localization in automated program repair," in *IEEE International Conference on Software Maintenance and Evolution*, 2017, pp. 504–508.

[3] 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*, pp. 1–1, 2019.

[4] W. Eric Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, pp. 1–1, 08 2016.

[5] 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.

[6] H. A. de Souza, M. L. Chaim, and F. Kon, "Spectrum-based software fault localization: A survey of techniques, advances, and challenges," *arXiv preprint arXiv:1607.04347*, 2016.

[7] 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)*, May 2018, pp. 433–444.

[8] 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*, 2014, pp. 643–653.

[9] A. Vahabzadeh, A. M. Fard, and A. Mesbah, "An empirical study of bugs in test code," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sep. 2015, pp. 101–110.

[10] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '05. New York, NY, USA: ACM, 2005, pp. 273–282.

[11] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, vol. 82, no. 11, pp. 1780 – 1792, 2009, sI: TAIC PART 2007 and MUTATION 2007.

[12] 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, 2014.

[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: ACM, 2019, pp. 101–111.

[14] A. Shi, J. Bell, and D. Marinov, "Mitigating the effects of flaky tests on mutation testing," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: ACM, 2019, pp. 112–122.

[15] M. Cordy, R. Rwemalika, M. Papadakis, and M. Harman, "Flakime: Laboratory-controlled test flakiness impact assessment. a case study on mutation testing and program repair," 2019.

[16] 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 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 437–440.

[17] X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 4, pp. 31:1–31:40, Oct. 2013.

[18] J. Sohn and S. Yoo, "Fluccs: Using code and change metrics to improve fault localization," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2017. New York, NY, USA: ACM, 2017, pp. 273–283.

[19] 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, March 2014.

[20] X. Xie, T. Y. Chen, F.-C. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 22, no. 4, p. 31, 2013.

[21] 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*, ser. ICSE '17. Piscataway, NJ, USA: IEEE Press, 2017, pp. 609–620.

[22] W. E. Wong, V. Debroy, Y. Li, and R. Gao, "Software fault localization using dstar (d*)," in *2012 IEEE Sixth International Conference on Software Security and Reliability*, 2012, pp. 21–30.

[23] R. Abreu, P. Zoeteweij, and A. J. c. Van Gemund, "An evaluation of similarity coefficients for software fault localization," in *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, Dec 2006, pp. 39–46.

[24] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund, "Spectrum-based multiple fault localization," in *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '09. IEEE Computer Society, 2009, pp. 88–99.

[25] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, Aug. 2011.

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

[27] F. Palomba and A. Zaidman, "The smell of fear: on the relation between test smells and flaky tests," *Empirical Software Engineering*, vol. 24, no. 5, pp. 2907–2946, Oct 2019.