Relationship between the Effectiveness of Spectrum-Based Fault Localization and Bug-Fix Types in JavaScript Programs

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

Attila Szatmári
Software Engineering Department
University of Szeged
Szeged, Hungary
szatma@inf.u-szeged.hu

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

***Abstract*—Spectrum-Based Fault Localization (SBFL) is a well-understood statistical approach to software fault localization, and there have been numerous studies performed that tackle its effectiveness. However, mostly Java and C/C++ programs have been addressed to date. We performed an empirical study on SBFL for JavaScript programs using a recent bug benchmark, BugsJS. In particular, we examined (1) how well some of the most popular SBFL algorithms, Tarantula, Ochiai and DStar, can predict the faulty source code elements in these JavaScript programs, (2) whether there is a significant difference between the effectiveness of the different SBFL algorithms, and (3) whether there is any relationship between the bug-fix types and the performance of SBFL methods. For the latter, we performed a manual classification of each benchmark bug according to an existing classification scheme. Results show that the performance of the SBFL algorithms is similar but there are some notable differences among them as well, and that certain bug-fix types can be significantly differentiated from the others (in both positive and negative direction) based on the fault localization effectiveness of the investigated algorithms.**

***Index Terms*—Spectrum-Based Fault Localization, JavaScript, bug classification, testing and debugging.**

I. Introduction

As part of the debugging process, software fault localization is an important activity in maintenance and evolution. Automated fault localization methods could help detect as many bugs as possible with the least effort and resources, hence given these methods, the efficiency of software development could increase. Spectrum-Based Fault Localization (SBFL) is a well-researched and well-understood class of automated fault localization methods [1], [2], [3], [4], [5]. The basic intuition behind SBFL is that those code elements (statements, blocks, functions, etc.) are more suspicious to contain a fault that are exercised by comparably more failing test cases than passing ones, while non-suspicious elements are traversed mostly by passing tests. Suspiciousness can be expressed in different ways, usually assigning one value to each code element (called the *suspiciousness score*), which can then be used to *rank* the code elements. When this ranked list is given to the developer for investigation, it is hoped that the fault will be found near the beginning of the list, hence providing useful advice.

However, the practical applicability of SBFL methods is still limited [6], [7], in fact, relatively bad performance in terms of ranking is one of the main reasons why it is not widespread [8], [9]. Other studies highlighted different barriers as well; applicability of theoretical results in practice [10], little experimental results with real faults [4], validity issues of empirical research [11], and not meeting practitioner's expectations [12]. Also, only a modest number of tools that employ SBFL techniques have been proposed over the years that are practically usable in real scenarios (GZoltar [13], Road2fault [14], iFL [15]).

Among the many possible causes of suboptimal performance of SBFL methods is how we understand the behavior of the algorithms given different types of bugs. In this area, only a few studies are avaialble [16], [17], [18], [19]. In this work, we present an empirical study to investigate the relationship between bug-fix types and fault localization efficiency. We used the JavaScript bug benchmark BugsJS [20] to compare how well the SBFL algorithms perform. Despite the fact that JavaScript is a popular programming language, automated fault localization in this language is less researched than for Java or C/C++, for instance. Hence, our work is a novel contribution in this regard as well.

Our main contributions in this paper are the following:

1) We implemented some of the well-known SBFL algorithms for JavaScript and applied them on the bug benchmark BugsJS.
2) We performed a categorization of bugs in the benchmark on function-level.
3) We empirically verified if the different SBFL algorithms behave significantly differently on the benchmark.
4) We empirically analyzed how the different bug-fix types relate to the fault localization effectiveness in terms of SBFL rank.

Results indicate that the performance of the three implemented SBFL algorithms is similar but there are some notable differences among them as well; Ochiai and DStar are more accurate than Tarantula, in particular. The other main finding

978-1-7281-5143-4/20 © 2020 IEEE

is that certain bug-fix types can be significantly separated from the others (in both positive and negative direction) based on the effectiveness of the three implemented SBFL algorithms. A possible implication of our findings is that future research on SBFL should investigate algorithms that consider specific bug types. Clearly, the analysis of *bug-fix* types is only indirectly relevant (the fix comes after localization), but this work can be extended to include other bug categorizations as well.

The rest of the paper is organized as follows. In the next section, we detail the goals of this study proposing two research questions. In Section III we give an overview of the overall study design, data preparation, bug labeling and computing the fault localization ranks. In Section IV, we examine these ranks and draw conclusions, thus answering the research questions. Finally, related work (Section V) and conclusions are offered (Section VI).

II. Goals and Research Questions

There were a few motivations for this work. First is the lack of previous research on automated fault localization for the JavaScript language. There are many fault localization approaches and their effectiveness has been analyzed by several publications, but these mainly concentrate on C/C++ and more recently on Java programs [1], [2], [3], [4], [5]. In spite of the fact that JavaScript has become an increasingly popular language recently[1], [2], only a few SBFL-related studies have been conducted on this language (e.g. [21], [22]). One of the reasons for this could be a lack of an accepted, validated benchmark for researchers. Recently, the bug benchmark BugsJS [20] has been published, so this enabled us to perform an empirical study of SBFL algorithms on JavaScript programs.

Thus, the goal of our research is to examine the performance of different SBFL algorithms using BugsJS. In particular, our goal was to find out if any of the most popular algorithms (Tarantula [23], Ochiai [24] and DStar [25]) produces better overall localization efficiency than the others. Furthermore, we were interested in the different types of bug fixes and if these affect the algorithms' fault localization efficiency. Simply by looking at how these algorithms compute the fault localization ranks (they look at code coverage details of passed and failed test cases), one could not easily predict if the bug (and the corresponding code fix) type could have any significant influence on the algorithms' performance. However, if we take into account that the developers have a tendency to make mistakes to various extent for different code constructs (more often in conditional statements than assignment expressions [26], for instance), a knowledge about how automated SBFL behaves in different situations could help design more specialized algorithms that could better serve the programmers.

In this paper, we are looking to answer the following Research Questions:

RQ1 Is any of Tarantula, Ochiai or DStar significantly different to the others in terms of fault localization efficiency for JavaScript bugs?

RQ2 Are there any bug-fix types on which any of these algorithms perform significantly differently (better or worse) than the others?

We expect that the answers to these questions could help researchers and developers to select and improve existing fault localization algorithms, and could also be beneficial in other related fields such as automated program repair [27], test generation [28], and bug prediction [29], among others.

III. Study Design and Data Preparation

A. Overview

Figure 1 contains the overview of our process to obtain empirical measurement data. We used the BugsJS benchmark suite [20], which was created to enable bug-related software engineering research with real, curated JavaScript bugs.[3] For each bug, BugsJS includes several related code revisions and sets of test cases, and enables individual execution of these versions. Execution information from related test cases can be obtained including per-test code coverage and test results. More information on how we treated the benchmark bugs is given in Section III-B.

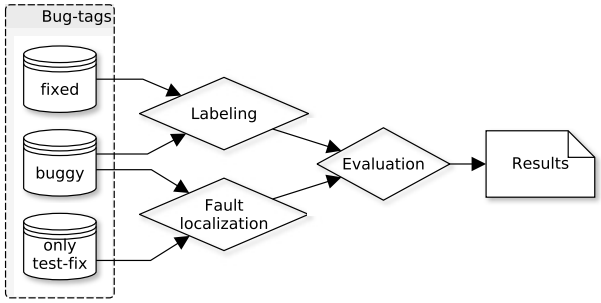

Fig. 1. Experiment overview

For the purposes of the present research, we used three code revisions ("bug-tags") for each bug in the benchmark:

- *buggy*: the parent commit of the revision in which the bug was fixed,
- *fixed* contains only the production code changes introduced to fix the bug, applied to the buggy revision, and
- *only test-fix* contains only the tests introduced in the bug-fixing commit, applied to the buggy revision.

Using this set of data, three steps were necessary to obtain the final results for answering our research questions:

1) Bug labeling, in which we calculated the function change sets between the buggy and fixed revisions for all bugs and determined the respective bug-fix labels.
2) Coverage measurement and fault localization score calculation, in which we collected coverage data and test results, and calculated the fault localization rank values for each source code element (function).
3) Data evaluation.

[1] http://pypl.github.io/PYPL.html

[2] https://insights.stackoverflow.com/survey/

[3] https://bugsjs.github.io/

Figure 2 shows the overall process of assigning bug-fix types to the benchmark bugs. Following the preliminary classification done by the benchmark authors [20], we started from the bug-fix types of Pan *et al.* [26]. Starting from the *buggy* and *fixed* code revisions, we used GitHub's *split diff* view to identify the changed code lines and the corresponding JavaScript functions with the help of the code coverage data. More information on the labeling process and the labels themselves are found in Section III-C.

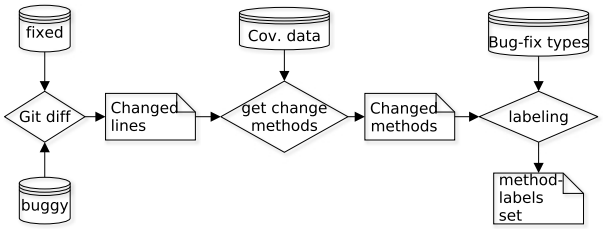

Fig. 2. Bug labeling process

Finally, Figure 3 shows how we calculated the fault localization results for the benchmark bugs. We used the "per-test" measurement feature of the benchmark that allows computing function-level code coverage data for each test case separately along with the test case outcomes. We ran the tests separately in the two tagged versions for each bug, *buggy* and *only test-fix*. We then compared the test results (pass or fail) of these two versions. We omitted those tests that failed in both cases, because these failed tests were most probably not related to the relevant bugs. The inputs to the SBFL algorithms were these filtered test results and the function-level code coverage data of the *buggy* versions. More details on the fault localization computation process is provided in Section III-D, while the procedure for evaluating the algorithms' fault localization efficiency is detailed in Section III-E.

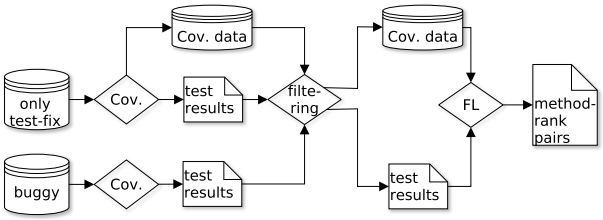

Fig. 3. Fault Localization process

B. Bug Dataset

BugsJS is a JavaScript bug dataset which includes 453 reproducible bugs from 10 popular (number of GitHub-stars ≥ 100), mature (number of commits ≥ 200) and active (year of the latest commit ≥ 2017) Node.js server-side JavaScript programs from GitHub that adopt the Mocha testing framework [20]. It consists of two other components: a Docker image that provides a runtime environment, and a bug dataset. The dataset can be queried through an API. The bugs are manually validated and cleaned to ensure that the bugs and their fixes were relevant (fixing commits are actually fixing the bug described in the issue), isolated (fixing code exclusively aim at fixing the bug, and no other changes are interleaved within it), and reproducible (the test cases were manually extracted that demonstrate each bug).

Although the current version of the BugsJS benchmark includes most of the functionality required to perform SBFL-related experiments off-the-shelf, some modifications had to be made to it. In particular, the "per-test" measurement executed for all bugs and all test cases is a very slow and space consuming process (for a big project like Eslint which has over 10 thousand test cases it took about a day to execute the "per-test" measurement for one bug). We added a new implementation of this feature that reduced the execution time by tenfold. The old implementation would instrument the same code, and write redundant information into the output files with every test run, and this was optimized in our version to process the data jointly for all test cases.

This modification had the consequence that some of the bugs present in the benchmark could not be used in our experiments. The reason was that there was a conflict between the Node.js version of new per-test coverage instrumentation and the version of buggy projects. In particular, we could use 7 out of 10 projects and 336 out of 453 bugs altogether in our experiments. Details about this are given in Table I, along with the average extent of the bugs in terms of code lines.

TABLE I
NUMBER OF BUGS USED IN THE STUDY

Project	Bower	Eslint	Express	Hessian.js	Hexo	Pencilblue	Shields
BugsJS	3	333	27	9	12	7	4
This study	3	282	25	8	12	3	3
Average number of tests	0.4K	12.4K	0.7K	0.1K	0.7K	0.4K	0.2K
Average number of methods	0.8K	1.9K	0.3K	0.1K	0.6K	0.3K	0.6K
Average number of modif. lines	12	7.2	8.5	12.5	6.8	7	7.1

C. Labeling

There may be various ways for the categorization of software bugs, and a lot of approaches have been published in this topic to day [26], [30], [31] In this research, we followed the approach to examine how the specific bugs have been *fixed* by the developers. This approach is much more robust and application independent than starting from the bug itself, and could be done in a relatively straightforward way given a well-defined classification scheme.

We started from the classification of Pan *et al.* [26] and, because it was designed for Java programs, we adapted it to JavaScript. We used only the highest level categories proposed in this work, and interpreted their meaning to JavaScript, which was straightforward in most of the cases. To our knowledge, there is no related work on labeling JavaScript

bug fixes. It was also not known what kinds of software bugs are the most common, and whether bug-fix types have similar frequency distributions across multiple systems.

Pan *et al.* defined the patterns by manually analyzing several open source Java projects (ArgoUML, Columba, Eclipse, JEdit and Scarab), and inspected the bugs and their fixes in the bug fix revision. Table II shows their final categorization.

TABLE II
CATEGORIES PROPOSED BY PAN ET AL.

Bug fix patterns	Description
IF	If-related: Changes to if conditions in any way.
MC	Method call: Changes that include calling a method.
SQ	Sequence: Three or four changes in an operation sequence.
LP	Loop: Change either in the loop predicate or in the scope of the loop statement.
AS	Assignment: The expression on the right hand side changes but the expression on the left stays the same.
SW	Switch: Additional or removal of switch branch
TY	Try-related: Additional or removal of a try or a catch block
MD	Method declaration: Addition or removal of a method declaration, or a change of method declaration
CF	Class field: Addition, removal or change of Class fields.

We will now describe the cases that match these patterns:

- The IF category of bug fix patterns from Table II includes changes made to the if statement. This can mean from *addition of a precondition check* to *removing an Else Branch*, and also changes made to an *if* predicate and adding or removing *else* or *else if* branches.
- MC includes cases that are related to method calls. This pattern matches cases such as change in the number of parameters, method call with different actual parameter values and change of method call to a class instance.
- SQ relates to additions or removals of operations in a field setting sequence or in a sequence of method calls to an object. This pattern also includes the case when there is addition or removal of method calls in a short construct body.
- LP is used whenever there is a change in a loop predicate or in an expression that modifies the loop variable inside the scope of the loop statement.
- AS is used for assignment related bug fixes. So, the bug fix changes the expression on the right hand side but the left hand side stays the same in both the buggy and the bug fix versions.
- SW is used whenever an addition or removal happens to a case in a switch statement.
- TY is used if the bug fix is try-related. This means that the bug fix adds a try-catch statement to enclose a section of code or removes a try-catch statement from the code. It can refer to addition or removal of the catch block as well.
- The MD pattern refers to the changes of method declarations. MD is intended to be used whenever there is an addition or removal of a method declaration.
- Label CF is used for the changes made to class fields. This can be addition or removal and the change of a class field declaration.

The most common bug fix patterns were IF and MC in Pan *et al.* 's study.

D. Computing Fault Localization Ranks

The details of the Spectrum-Based Fault Localization algorithms we implemented for this work are the following.

1) Coverage and Result Data: To store the program execution information, we calculated a *coverage matrix* (the program spectrum), whose rows represent the tests (T), and columns show the (source) code elements (for example, functions - F). The 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. With another data structure (the *result vector*) the results of the tests are stored, where 0 means that test τ passed, otherwise it is 1.

$$COV = \begin{bmatrix} c_{1,1} & \dots & c_{1,a} \\ \vdots & \ddots & \vdots \\ c_{b,1} & \dots & c_{b,a} \end{bmatrix}, \; RES = \begin{bmatrix} r_1 \\ \vdots \\ r_b \end{bmatrix},$$

$$c_{\tau,\phi} \in \{0,1\} : \; \tau \in T \; \phi \in F,$$
$$c_{\tau,\phi} = 1 \text{ if } \phi \text{ is covered by } \tau,$$

$$r_\tau \in \{0,1\} : \; \tau \in T,$$
$$r_\tau = 0 \text{ if } \tau \text{ is pass, else } 1.$$

2) Metrics: Given the spectrum above, four basic statistics (metrics) are calculated for a function ϕ:

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

d) ϕ_{nf}: number of failing tests not covered by ϕ

These four values are then used by many formulae [7], [32] to compute the corresponding suspciousness scores of ϕ, and thus providing a ranked list of code elements as the final output. There have been numerous formulae proposed in literature [1], [2], [3], [4], [5], but some of the most often used ones are Tarantula [23], Ochiai [24] and DStar [25], so we implemented these to be used in our experiments and are given below:

DStar [25] : $$\frac{\phi_{ef}^2}{\phi_{ep} + (\phi_{ef} + \phi_{nf}) - \phi_{ef}},$$

Ochiai [24]: $$\frac{\phi_{ef}}{\sqrt{(\phi_{ef} + \phi_{nf}) \cdot (\phi_{ef} + \phi_{ep})}},$$

Tarantula [23]: $$\frac{\frac{\phi_{ef}}{\phi_{ef}+\phi_{nf}}}{\frac{\phi_{ef}}{\phi_{ef}+\phi_{nf}} + \frac{\phi_{ep}}{\phi_{ep}+\phi_{np}}}.$$

E. SBFL Algorithm Evaluation Process

To evaluate (and quantify) the effectiveness of SBFL algorithms and the relationship between them and bug-fix types, we will use the position of the buggy functions in the suspiciousness rank order provided by the algorithms. This is preferable over comparing the suspiciousness scores directly as the different algorithms produce the scores on different scales or produce different distributions.

In a simplest form, a bug-rank is the absolute position of changed (buggy) method in suspiciousness order [33], but often it is examined in a relative form with respect to the total number of code elements. This approach makes it more comparable among different subject programs, however recent user studies report that developers tend to investigate only the top 5 or at most the top 10 elements in the recommendation list provided by localization methods before giving up [8], [12]. Hence, any improved rank position which is above these thresholds will probably be less useful, no matter how much relative improvement we can achieve. The importance of absolute measures was highlighted by Parnin and Orso as well [9]. Therefore we divided the rank-scale into five special segments (in a similar fashion to the approach used by Zou *et al.* [34]). We will distinguish between cases where the code element has a rank equal to 1 (*top-1*), it is less or equal to three (*top-3*), less or equal to five (*top-5*), less or equal to ten (*top-10*), and when it is over ten (*other*), commonly referred to as *top-N*.

Another topic when expressing the efficiency of fault localization is the handling of ties [35], because in many cases it can happen that different program elements get assigned the same suspiciousness scores. There are three commonly used approaches for comparing rankings when many elements have the same ranks: (1) average of the ranks: these methods will get their average rank, (2) minimum rank: the lowest rank will be assigned to the methods and (3) maximum rank: the methods take the highest rank value. In this work, we follow the approach to examine average ranks. Finally, fixing a bug can affect more code elements (functions, in our case), and to handle this situation we used the lowest value of the affected functions' ranks. In the examined programs, there were 272 bugs where the patch only affected one function and 64 changes that resulted in changes to two or more functions.

IV. Results

In this section, we will present the results of comparing the fault localization data based on the described effectiveness measure, first answering RQ1 (Section IV-A), and then RQ2 (Section IV-B). We will conclude this section with general discussion of the results and threats to validity.

A. Overall Ranks

Table III shows the average ranks obtained per project and per algorithm. First, we can observe that the three approaches typically produced similar results, apart from some outlier cases, most notably, DStar on the Hexo project. This is not so surprising as other studies have already found that this algorithm can produce extreme ranks in some cases [36]. It can be stated that, overall, Ochiai obtained the best results, i.e. the lowest average ranks. It is interesting to note that there were no cases where the average rank of Tarantula was lower than that of Ochiai, but surprisingly DStar gave worse results than Ochiai only on Hexo and in all other cases it was better. Without the outlier, DStar would have been the clear winner among the three methods. Other works that worked with these algorithms but on different languages, obtained similar results [4], [25], [37], [38].

TABLE III
Average ranks

Project	Tarantula	Ochiai	DStar
Bower	25.83	19.17	17.50
Shields	5.83	5.83	5.17
Hexo	3.25	3.00	80.88
Hessian.js	4.81	3.88	3.00
Express	8.10	7.94	7.94
Pencilblue	1.83	1.67	1.67
Eslint	20.39	19.90	19.90
All	18.24	17.73	20.47

To verify if there is a statistically significant difference between the algorithms, we used the Wilcoxon signed rank test on the fault localization ranks [39]. This is a non-parametric statistical hypothesis test, which is used to compare two samples to assess whether their population mean ranks differ. We applied it to determine whether these two dependent samples were selected from populations having the same distribution. In particular, we wanted to know if any of the three algorithms is significantly better than the others. Significance level was chosen to be $\alpha = 0.05$, and we set the null hypothesis (H_0) to be that a fault localization algorithm A is not significantly better than algorithm B.

Wilcoxon signed rank test gave us a $p = 0.00001$ value for the Tarantula-Ochiai pair, so we reject H_0 and accept H_1, that is, there is a difference in efficiency between the two algorithms. Same goes for Tarantula and Dstar, the test gave us the value of $p = 0.00043$. For the Ochiai - DStar pair, the p value was bigger than α ($p = 0.70975$), therefore H_0 was adopted in this case, that is, the two approaches produce similar results (despite DStar's some extreme rankings).

The average of the ranks can be easily misleading as high average rank values can be caused by some extremely bad values and they can influence the overall results greatly (such as the case with DStar and Hexo). So, we investigated also a set of rank positions which we believe are particularly important. As mentioned earlier, the practical usability of SBFL depends on the position of the faulty element (in absolute terms, not in relative), and that developers tend to investigate only the top 5 or at most the top 10 elements. We used the top-N ranking scheme in this set of experiments, as introduced in Section III-E.

The top-N values for all the bugs are shown in Table IV. Overall, almost 30% of bugs have a rank of 1 and 90% of them have ranks of 10 or less. It can be observed that Ochiai and DStar continue to produce similar results (and this is especially true for the top-1, top-3 and top-5 categories). This also indicates that DStar is not performing as bad as the average rank showed above, which was due to an outlier.

TABLE IV
TOP-N RANKS

	Tar.	(%)	Ochiai	(%)	DStar	(%)
top-1	96	28.6	101	30.1	100	29.8
top-3	206	61.3	214	63.7	214	63.7
top-5	255	75.9	264	78.6	264	78.6
top-10	300	89.3	306	91.1	303	90.2
other	36	10.7	30	8.9	33	9.8

Figures 4–6 show this data in a slightly different manner. Here, we did not use the top-N values as defined above, but their non-accumulating variant. This means that we counted the cases where a particular bug fell into a non-overlapping interval of $[1]$, $(1,3]$, $(3,5]$, $(5,10]$ or $(10,\dots]$. We see from Figure 4, for example, that there are 48 bugs that rank between 3 and 5 (right closed) interval based on Tarantula.

Overall, ≈30% of bugs belong to the best $[1]$ category, but the most elements are found in interval $(1,3]$ (nearly 33%). It can be observed that each algorithm produces similar results, however, Ochiai and DStar contain more elements in the segments $[1]$, $(1,3]$, $(3,5]$ than Tarantula. In addition, Tarantula assigned most of the elements to the two worst groups: $(5,10]$: 44 functions, 13.4% and $(10,\dots]$: 36 functions, 10.7%. This information confirms our earlier findings based on average ranks (Table III) that the results are similar, but Tarantula slightly lags behind.

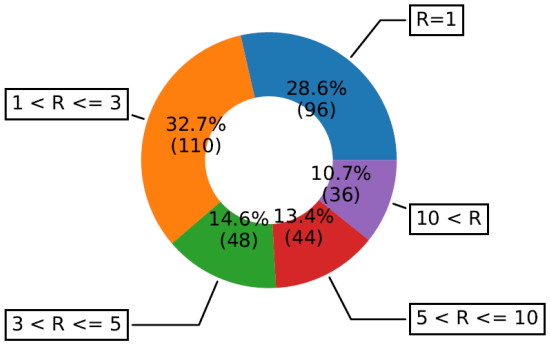

Fig. 4. Tarantula ranks

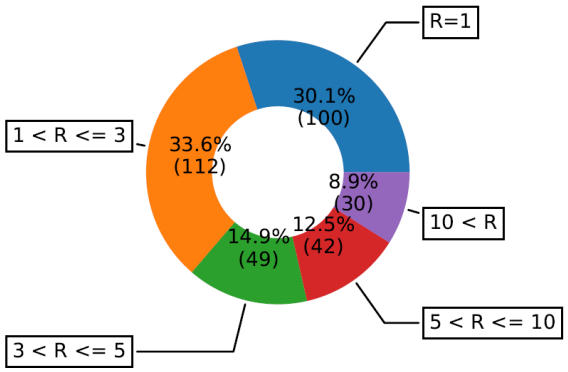

Fig. 5. Ochiai ranks

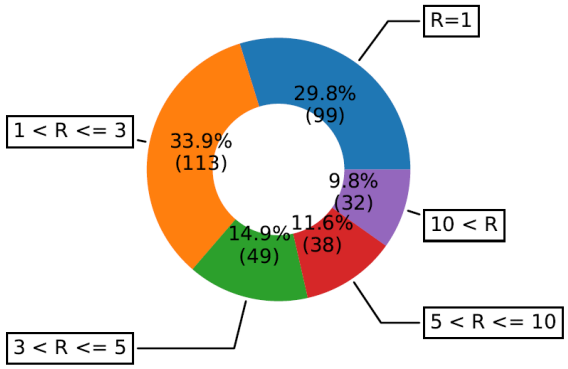

Fig. 6. DStar ranks

RQ 1: Concluding this research question, we found that there is no significant difference between the Ochiai and DStar fault localization algorithms, but Tarantula is different (slightly worse) than any of the other two. Both average rankings (Table III) and the results of the top-N analysis (Table IV and Figures 4–6) support this conclusion. In terms of the top-N positions, in about 30% of the cases perfect localization could be achieved, and in about 90% of the cases the fault was among the top-10 positions.

B. Ranks of Different Bug-Fix Types

Based on the bug-fix categories discussed in Section III-C, we assigned labels to each of the JavaScript functions participating in the investigated bugs. Table V shows the associated statistics for Tarantula: for each bug category and subject system, we give the number of functions that got assigned that category.

Note that the number of labels (412) is greater than the number of bugs (336). This is because there are several functions that have multiple tags associated with them. The three algorithms produced very similar results (hence we give numbers only for one), with one difference (and it is marked with $*$ in Table V): in the ESlint project, 75 of the modified

functions have AS bug-fix type based on Tarantula, but it is 76 based on Ochiai and DStar. This may occur because it is not guaranteed that all fault localization algorithms will be assigned the lowest rank for the same method and this can cause the labels to differ. This difference also affects aggregate values, and these are also marked in the table.

The numbers of IF (176) and AS (102 or 103) appear to be prominent and MC (65) is quite high as well, but there are also very rare types (e.g. SW - 5 or TY - 1).

TABLE V
OVERALL BUG-FIX TYPE STATISTICS BASED ON TARANTULA

Bug-fix type	Bower	Shields	Hexo	Hessian	Express	Pencilblue	Eslint	Total
IF	2	0	5	4	12	2	151	176
AS	2	3	9	2	9	2	75*	102*
MD	0	0	0	0	1	0	23	24
LP	0	0	0	0	2	0	6	8
MC	0	0	1	0	3	1	60	65
SQ	0	0	1	3	6	0	19	29
SW	0	0	0	0	1	0	4	5
TY	0	0	0	0	0	0	1	1
CF	0	1	0	0	0	0	1	2
Total	4	4	16	9	34	5	340*	

The goal of the experiments presented in this section was to find out if there are any bug-fix types for which the SBFL algorithms' efficiency is significantly different (either better or worse) than for the others. Similarly to the overall rank analysis from the previous section, we counted the number of bugs belonging to the different top-N categories and separated them according to the bug-labels. That is, we wanted to see what kinds of labels were assigned to the lowest ranked (modified) functions for each bug, and how they were distributed among the top-N categories. If more than one function was associated with a bug only the lowest rank (and labels) were considered, and if there were more than one label associated with a function then that function was accounted for each label.

Table VI shows the distribution of the labels obtained for the three algorithms by category. An element in the table tells us how many bugs there are where the least-ranked modified function contains the given bug-fix type and the rank falls within a given top-N range. For example, there are 115 bugs where the lowest Tarantula rank is 3 or less and this function has an IF bug-fix label.

The relative versions of these numbers are provided in Table VII. Here, we can see what percentage of items with the specified label are in the top-N category. For example, 65.3% of least-ranked modified functions with the IF tag have a rank less or equal to 3. When analyzing the most common labels, it is interesting to note that IF is slightly better, and AS is slightly worse than the top-N results in Table IV. Also, the results of SQ are interesting because the number (and proportion) of labels in the top-1 category is very low while its *other* category is high. Other labels produced low element numbers, so their general interpretation is less relevant.

As in the previous section, we counted the number of labels occurring in the non-overlapping rank-ranges as well. The associated results can be seen in Figures 7–9. The more common labels (IF and AS) also show that there are nearly as many labels in $[1]$ interval as labels with a value rank of 2 to 3. In addition, Tarantula is a bit less efficient at finding buggy functions than Ochiai or DStar. For example, the number of labels in $(1, 3]$ is 131 for Tarantula, 136 for Ochiai and 141 for DStar. We can again see that the occurrence of SQ labels in the lower ranges is quite low compared to the others, while IF is the opposite: it has a high proportion of labels in the top categories and fewer in the worse rank positions.

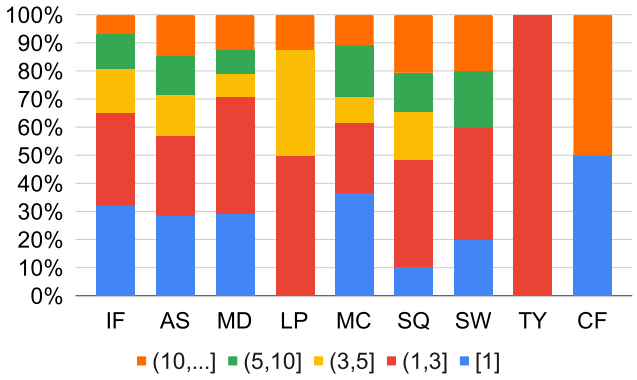

Fig. 7. Tarantula interval statistics

Figure 7 shows that there are certain bug types which Tarantula finds easier. Although this can be misleading if there is not enough data, e.g. on TY Tarantula performs really well, however, as Table VI shows there is only one bug-fix with this label.

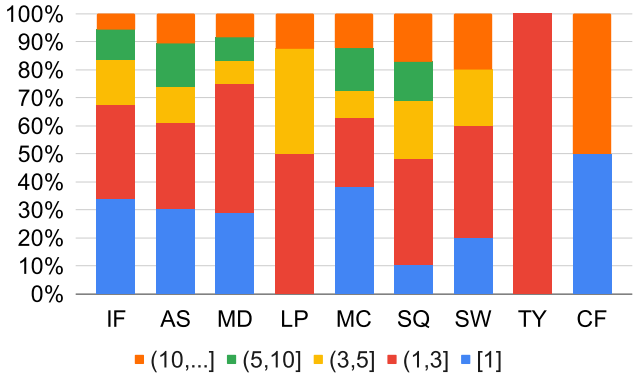

Fig. 8. Ochiai interval statistics

As we can see in Figures 7, 8 and 9, the labels IF, AS, MD in the [1] interval have similar results as R=1 (Top 1) in Figure 4, 5 and 6.

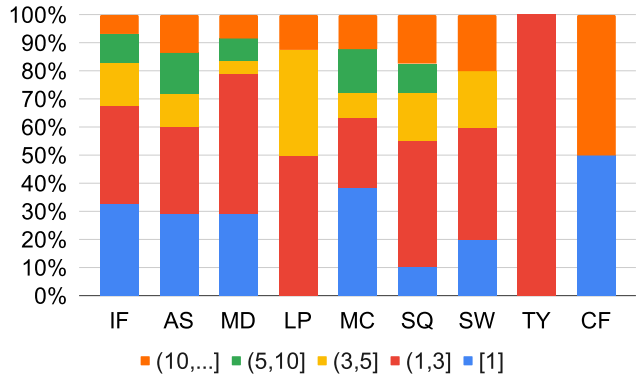

Fig. 9. DStar interval statistics

To verify if this data shows any statistically significant trends, we used Fisher's exact test. It is a statistical significance test, which is one of the non-parametric methods and is used in the analysis of contingency tables [40]. We counted the number of labels per metrics provided by the three SBFL algorithms in Table VI, and in order to perform the test, we created the contingency tables for each *(bug-fix types, non-overlapping interval, algorithm)* configuration. A contingency template is shown in Table VIII, where α is an algorithm, ω is a bug-fix type and γ is a top-N category. The values in the table cells indicate different counts of buggy functions:

a: which have label ω and their rank is in the range,
b: whose label set does not contain ω and their rank is in the range,
c: which have label ω and their rank is not in the range, and
d: which have a set of labels not containing ω and their rank is not in the range.

For example, when we choose the *(Tarantula, top-5, IF)* configuration, a = 142 (top-5 and IF), b = 169 (top-5 and not-IF), c = 34 (not-top-5 and IF) and d = 67 (non-top-5 and not IF) in the template from Table VIII, which results in $p = 0.037$ (see Tables VI and IX).

TABLE VIII
FISHER EXACT TEST (TEMPLATE)

α	ω	$\neg\omega$
γ	a	b
$\neg\gamma$	c	d

Table IX shows the p values that the labels get in the different intervals besides the given algorithm for the Fisher's exact test. The null hypothesis is that the ratio of belonging to a top-N range is not higher for one label than for others. If this value is less than our chosen significance level, 0.05, then we reject the null hypothesis and we can say that the ratio of belonging to a range is different (higher or lower) for a given label than the others, *i.e.* the proportion of the labels in the range is different. This test only determines whether there is a difference in probability, it does not determine its direction.

In Table IX, we highlighted the cases where the difference was significant according to the test. We can make the following observations based on this data. First, in the top-1 category, SQ is significantly different (worse) with all algorithms (this type seems to be more difficult to be localized), which we observed from previous data as well. In the top-5, top-10 and other categories the label IF is significantly different than the other labels; in top-5 and top-10 they are likely to be found by the SBFL algorithms. Also, in the *other* section it is significantly different in the opposite direction (there are less labels here). Table VII can be used to determine the direction of significance: for SQ, the number of elements in the top-1 category/section is low as opposed to IF where its ratio is high.

RQ 2: There are two bug-fix types that are significantly different from the others in terms of how successful the SBFL algorithms can locate them. Faults that require modifications of SQ type are less likely to be successfully localized at very high rank positions by the algorithms, while faults belonging to the IF category are ranked higher than other types in the top-5 and top-10 ranges.

C. Discussion

Our overall findings related to the different SBFL algorithms on JavaScript are not surprising: they behave in a similar way, but differences could be observed as well, which are aligned to previous studies. The fact that Tarantula was one of the earliest published SBFL methods, which was then subsequently challenged by a number of other approaches, is reflected in its worst overall performance (related to RQ1). Also, DStar produced a bit more hectic behavior (with its outlier result), which confirms other studies. But excluding this, it showed the overall best results. We conclude that we cannot claim a clear winner among these algorithms, similar to most of earlier studies performed for other languages. We suggest for future studies of SBFL on JavaScript programs to continue using multiple approaches, rather than staying with only one.

It was not among the goals of this paper to analyze the possible causes of our findings. We did not compare the overall fault localization effectiveness obtained for JavaScript bugs to similar results published for other languages, hence this is an interesting possibility for future work. Furthermore, it is not currently known why is it harder to localize SQ type bugs and easier for the IF bugs, as indicated by our findings related to RQ2. We can speculate that IF type of bug-fixing often alter the control flow of the program and hence cause different test cases to be activated which contributes to more diverse statistics for the four basic metrics in SBFL. On the other hand, SQ type of bug fixes merely change some sequential computation and perhaps these are less likely to cause different test cases to be triggered and behave differently. This area of research is for future work as well.

TABLE VI
NUMBER OF LABELS (PER METRICS)

Name	Top-1 (#)			Top-3 (#)			Top-5 (#)			Top-10 (#)			Other (#)		
	Tarantula	Ochiai	DStar	Tarantula	Ochiai	DStar	Tarantula	Ochiai	DStar	Tarantula	Ochiai	DStar	Tarantula	Ochiai	DStar
IF	57	60	58	115	119	119	142	147	146	164	166	164	12	10	12
AS	29	31	30	58	63	62	73	76	74	87	92	89	15	11	14
CF	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
MD	7	7	7	17	18	19	19	20	20	21	22	22	3	2	2
MC	24	25	25	40	41	41	46	47	47	58	57	59	7	8	8
SQ	3	3	3	14	14	16	19	20	21	23	24	24	6	5	5
SW	1	1	1	3	3	3	3	4	4	4	4	4	1	1	1
LP	0	0	0	4	4	4	7	7	7	7	7	7	1	1	1
TY	0	0	0	1	1	1	1	1	1	1	1	1	0	0	0
All	122	128	125	253	264	266	311	323	321	366	374	369	46	39	44

TABLE VII
PERCENTS OF LABELS (PER METRICS)

Name	Top-1 (%)			Top-3 (%)			Top-5 (%)			Top-10 (%)			Other (%)		
	Tarantula	Ochiai	DStar	Tarantula	Ochiai	DStar	Tarantula	Ochiai	DStar	Tarantula	Ochiai	DStar	Tarantula	Ochiai	DStar
IF	32.4	34.1	33.0	65.3	67.6	67.6	80.7	83.5	83.0	93.2	94.3	93.2	6.8	5.7	6.8
AS	28.4	30.1	29.1	56.9	61.2	60.2	71.6	73.8	71.8	85.3	89.3	86.4	14.7	10.7	13.6
CF	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0	50.0
MD	29.2	29.2	29.2	70.8	75.0	79.2	79.2	83.3	83.3	87.5	91.7	91.7	12.5	8.3	8.3
MC	36.9	38.5	38.5	61.5	63.1	63.1	70.8	72.3	72.3	89.2	87.7	87.7	10.8	12.3	12.3
SQ	10.3	10.3	10.3	48.3	48.3	55.2	65.5	69.0	72.4	79.3	82.8	82.8	20.7	17.2	17.2
SW	20.0	20.0	20.0	60.0	60.0	60.0	60.0	80.0	80.0	80.0	80.0	80.0	20.0	20.0	20.0
LP	0.0	0.0	0.0	50.0	50.0	50.0	87.5	87.5	87.5	87.5	87.5	87.5	12.5	12.5	12.5
TY	0.0	0.0	0.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	100.0	0.0	0.0	0.0

TABLE IX
SIGNIFICANCE IN TOP-N BASED ON FISHER EXACT TEST

Name	top-1			top-3			top-5			top-10			other		
	Tarantula	Ochiai	DStar	Tarantula	Ochiai	DStar	Tarantula	Ochiai	DStar	Tarantula	Ochiai	DStar	Tarantula	Ochiai	DStar
IF	0.326	0.237	0.330	0.406	0.214	0.254	**0.037**	**0.030**	**0.031**	**0.017**	**0.027**	**0.036**	**0.017**	**0.027**	**0.036**
AS	0.804	0.902	0.806	0.157	0.554	0.342	0.291	0.217	0.103	0.206	0.697	0.272	0.206	0.697	0.272
MD	1.000	1.000	1.000	0.516	0.281	0.131	0.809	0.799	0.619	0.741	1.000	1.000	0.741	1.000	1.000
LP	0.112	0.113	0.113	0.471	0.467	0.463	0.686	1.000	0.691	1.000	0.551	0.597	1.000	0.551	0.597
MC	0.183	0.145	0.141	0.780	0.889	0.888	0.348	0.251	0.258	1.000	0.362	0.661	1.000	0.362	0.661
SQ	**0.019**	**0.012**	**0.019**	0.109	0.074	0.316	0.261	0.242	0.489	0.118	0.176	0.218	0.118	0.176	0.218
SW	1.000	1.000	1.000	1.000	1.000	1.000	0.600	1.000	1.000	0.448	0.393	0.432	0.448	0.393	0.432
TY	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000	1.000
CF	0.505	0.519	0.514	1.000	1.000	1.000	0.431	0.389	0.396	0.211	0.180	0.202	0.211	0.180	0.202

D. Threats to Validity

We identified the following possible threats to the validity of our empirical study, for which we also list our attempts to mitigate them.

The BugsJS benchmark is relatively new and hence no comparison basis is available to previous work using it. However, given that the bugs are diverse and manually validated by the benchmark authors, and that it is very similar to other benchmarks used in related research (e.g., Defects4J [41]), we have confidence in this dataset. Similarly, the bug labeling was done in a similar fashion to other bug benchmarks, and in addition, all authors participated in this manual activity to counter for personal bias. Some of the bug-fix types had very few instances, so no strong conclusions should be drawn for these. However, we used statistical significance test to make sure that the main findings are not by chance.

We deliberately did not choose only one specific SBFL algorithm to be used in our experiments, to make sure that the results are independent of the actual algorithm.

V. Related Work

A. Fault Localization

SBFL has a large literature, and there are excellent surveys that overview the field [1], [2], [3], [4], [5].

Renieris and Reiss [42] presented a method which is based on the differences between pass and failed test. Jones and Harrold introduced the Tarantula fault localization metric [23] and showed that this approach outperforms the method by Renieris and Reiss on C programs. Abreu et al. used the Ochiai method in their studies [24], [37]. It was shown that Ochiai produces better results than Tarantula using the Siemens and the SIR [4] bug dataset. Wong et al. [25] presented the DStar technique, which was evaluated in 24 programs and the results were compared with 38 different techniques. Single-fault and multi-fault programs were used for assessment.

Pearson et al. [4] evaluated fault localization techniques and examined them to find whichever technique is the best for real faults. They used the bug benchmark Defects4J [41] to evaluate the algorithms.

There are many studies [37], [38], [25], [4] that compare the results of different fault localization algorithms. Some of the common conclusions of these studies are that (1) there is a difference in the efficiency between injected and real bugs, (2) Ochiai performed better than Tarantula (3), and DStar was better than Ochiai. The results of our investigation confirmed the latter two findings.

Regarding the different languages, Lucia et al [19] used Tarantula and Ochiai techniques to understand how well they perform on programs written in C as compared to programs written in Java (they found no significant difference). But, to our knowledge, SBFL was not investigated earlier for JavaScript programs.

B. Labeling

Many studies dealt with bug fixes, and they have been investigated from different aspects. Yin et al. [43] state that humans are prone to making errors in fixes which lead to bugs. This study showed that concurrency bugs are the most difficult ones to fix, and the authors also defined three patterns: memory bug, concurrency bug and semantic bug. Osman et al. [44] collected Java projects from GitHub. They analysed the change histories by linking revisions to bug fixes. They compared the two versions of the methods, i.e. the version before the fix and the version after the fix. Due to the large number of diversity of the analyzed projects, they decided to include only a few patterns in their study: a) Missing null checks, b) Missing Invocation, c) Wrong Name, d) Undue Invocation.

Lucia et al. [19] performed a similar study to ours. They, however, used C and Java projects. They divided bugs into several groups based on the bug-fix categories proposed by Pan et al. [26], and also added new bug categories such as CH-RET, OTH. They measured the effectiveness of each bug category by the proportion of bugs localized. They came to the conclusion that Ochiai could better localize bugs in CH-NCS (Addition/removal of non-conditional statements) than others.

Hanam et al. [30] and Ocariza et al. [45] classified JavaScript bugs and investigated their root causes. Hanam et al. did an empirical study on labeling JavaScript serverside bugs, and they proposed BugAID, a data mining technique for discovering common unknown bug patterns. They showed that language construct changes are closely related to bug patterns.

Martinez et al. [46] proposed another tool for mining change pattern instances from Git commits, and identifed ten change patterns, after analyzing the bug benchmark Defects4J.

[4]https://sir.csc.ncsu.edu/portal/index.php

VI. Conclusion

In this paper, we presented an empirical study of Spectrum Based Fault Localization for JavaScript programs. To our knowledge, this is the first systematic study of this class of algorithms for this language, and also the first related application of the recently published bug benchmark, BugsJS. We used three traditional SBFL algorithms in our experiments and found that they produce similar results to each other, but differences could also be observed that are aligned with previous studies for other languages.

Another goal of our research was to find out if there were differences in fault localization effectiveness between different types of bugs based on the associated bug fixes. We found that there are certain bug-fix types that seem to be harder to localize by the current algorithms (for instance, operation sequence change), while some others are easier than the rest (*if* condition related bugs). The analysis of bug-fix types is a first step and will be extended to include other bug categorizations that include bug causes as well, and hence contribute more directly to designing better fault localization algorithms.

Investigating the possible explanations for these phenomena are among our future work. Once we understand the underlying causes of the different behavior of the algorithms on different bug-fix types (and eventually other bug categories), we may be able to design improved algorithms in place of these very general ones we use currently, which would perform better on multiple types of bugs. Other possible implications of our research results include useful insights to researchers working in related fields such as automated program repair, test generation, and bug prediction.

Acknowledgment

Szatmári was supported by project EFOP-3.6.3-VEKOP-16-2017-0002, co-funded by the European Social Fund. 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] 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.

[2] P. Parmar and M. Patel, "Software fault localization: A survey," *International Journal of Computer Applications*, vol. 154, no. 9, 2016.

[3] H. A. de Souza, M. L. Chaim, and F. Kon, "Spectrum-based software fault localization: A survey of techniques, advances, and challenges," *ArXiv*, vol. abs/1607.04347, 2016.

[4] 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*. IEEE Press, 2017, pp. 609–620.

[5] P. Agarwal and A. P. Agrawal, "Fault-localization techniques for software systems: A literature review," *SIGSOFT Softw. Eng. Notes*, vol. 39, no. 5, pp. 1–8.

[6] F. Keller, L. Grunske, S. Heiden, A. Filieri, A. van Hoorn, and D. Lo, "A critical evaluation of spectrum-based fault localization techniques on a large-scale software system," in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. IEEE, 2017, pp. 114–125.

[7] 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.

[8] X. Xia, L. Bao, D. Lo, and S. Li, ""Automated Debugging Considered Harmful" Considered Harmful: A User Study Revisiting the Usefulness of Spectra-Based Fault Localization Techniques with Professionals Using Real Bugs from Large Systems," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, oct 2016, pp. 267–278.

[9] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ser. ISSTA '11. New York, NY, USA: ACM, 2011, pp. 199–209.

[10] T.-D. B. Le, F. Thung, and D. Lo, "Theory and Practice, Do They Match? A Case with Spectrum-Based Fault Localization," in *2013 IEEE International Conference on Software Maintenance*. IEEE, sep 2013, pp. 380–383.

[11] F. Steimann, M. Frenkel, and R. Abreu, "Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ser. ISSTA 2013. New York, NY, USA: ACM, 2013, pp. 314–324.

[12] P. S. Kochhar, X. Xia, D. Lo, and S. Li, "Practitioners' expectations on automated fault localization," in *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*. New York, New York, USA: ACM Press, 2016, pp. 165–176.

[13] 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.

[14] H. A. de Souza, D. Mutti, M. L. Chaim, and F. Kon, "Contextualizing spectrum-based fault localization," *Information and Software Technology*, vol. 94, pp. 245 – 261, 2018.

[15] G. Balogh, V. Schnepper Lacerda, F. Horváth, and Á. Beszédes, "iFL for Eclipse—a tool to support interactive fault localization in Eclipse IDE," in *Proceedings of 12th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2019.

[16] W. Masri, R. Abou-Assi, M. El-Ghali, and N. Al-Fatairi, "An empirical study of the factors that reduce the effectiveness of coverage-based fault localization," in *Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*. ACM, 2009, pp. 1–5.

[17] B. Baudry, F. Fleurey, and Y. Le Traon, "Improving test suites for efficient fault localization," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 82–91.

[18] W. Masri and R. A. Assi, "Prevalence of coincidental correctness and mitigation of its impact on fault localization," *ACM transactions on software engineering and methodology (TOSEM)*, vol. 23, no. 1, p. 8, 2014.

[19] L. Lucia, D. Lo, L. Jiang, F. Thung, and A. Budi, "Extended comprehensive study of association measures for fault localization," *J. Softw. Evol. Process*, vol. 26, no. 2, pp. 172–219, Feb. 2014.

[20] P. Gyimesi, B. Vancsics, A. Stocco, D. Mazinanian, Á. Beszédes, R. Ferenc, and A. Mesbah, "BugsJS: a benchmark of JavaScript bugs," in *Proceedings of the 12th IEEE Conference on Software Testing, Verification and Validation (ICST'19)*, Apr. 2019, pp. 90–101.

[21] F. S. Ocariza Jr, G. Li, K. Pattabiraman, and A. Mesbah, "Automatic fault localization for client-side javascript," *Software Testing, Verification and Reliability*, vol. 26, no. 1, pp. 69–88, 2016.

[22] F. Ocariza, K. Bajaj, K. Pattabiraman, and A. Mesbah, "An empirical study of client-side javascript bugs," in *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2013, pp. 55–64.

[23] 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.

[24] 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.

[25] 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.

[26] K. Pan, S. Kim, and E. J. Whitehead, Jr., "Toward an understanding of bug fix patterns," *Empirical Softw. Engg.*, vol. 14, no. 3, pp. 286–315, Jun. 2009.

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

[28] J. M. Rojas, M. Vivanti, A. Arcuri, and G. Fraser, "A detailed investigation of the effectiveness of whole test suite generation," *Empirical Software Engineering*, pp. 1–42, 2016.

[29] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," 05 2010, pp. 31–41.

[30] Q. Hanam, F. S. d. M. Brito, and A. Mesbah, "Discovering bug patterns in JavaScript," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: ACM, 2016, pp. 144–156.

[31] A. Vahabzadeh, A. M. Fard, and A. Mesbah, "An empirical study of bugs in test code," in *Proceedings of the 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, ser. ICSME '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 101–110.

[32] 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.

[33] 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 (TAICPART-MUTATION 2007)*. IEEE, 2007, pp. 89–98.

[34] 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.

[35] X. Xu, V. Debroy, W. E. Wong, and D. Guo, "Ties within fault localization rankings: Exposing and addressing the problem," *International Journal of Software Engineering and Knowledge Engineering*, vol. 21, pp. 803–827, 2011.

[36] B. Vancsics, "NFL: Neighbor-based fault localization technique," in *2019 IEEE 1st International Workshop on Intelligent Bug Fixing (IBF)*, Feb 2019, pp. 17–22.

[37] 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.

[38] 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.

[39] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[40] A. Agresti *et al.*, "A survey of exact inference for contingency tables," *Statistical science*, vol. 7, no. 1, pp. 131–153, 1992.

[41] 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. ACM, 2014, pp. 437–440.

[42] M. Renieres and S. P. Reiss, "Fault localization with nearest neighbor queries," in *18th IEEE International Conference on Automated Software Engineering, 2003. Proceedings.*, Oct 2003, pp. 30–39.

[43] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram, "How do fixes become bugs?" in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 26–36.

[44] H. Osman, M. Lungu, and O. Nierstrasz, "Mining frequent bug-fix code changes," *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pp. 343–347, 2014.

[45] F. S. Ocariza Jr., K. Pattabiraman, and B. Zorn, "Javascript errors in the wild: An empirical study," in *2011 IEEE 22nd International Symposium on Software Reliability Engineering*, Nov 2011, pp. 100–109.

[46] M. Martinez and M. Monperrus, "Coming: A tool for mining change pattern instances from git commits," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, May 2019, pp. 79–82.