A Critical Evaluation of Spectrum-Based Fault Localization Techniques on a Large-Scale Software System

Fabian Keller*, Lars Grunske†, Simon Heiden†, Antonio Filieri‡, Andre van Hoorn*, Lucia¶, and David Lo§
*University of Stuttgart, †Humboldt-Universität zu Berlin, ‡Imperial College London
¶University of Luxembourg, §Singapore Management University

***Abstract*—In the past, spectrum-based fault localization (SBFL) techniques have been developed to pinpoint a fault location in a program given a set of failing and successful test executions. Most of the algorithms use similarity coefficients and have only been evaluated on established but small benchmark programs from the Software-artifact Infrastructure Repository. In this paper, we evaluate the feasibility of applying 33 state-of-the-art SBFL techniques to a large real-world project, namely ASPECTJ. From an initial set of 350 faulty version from the iBugs repository of ASPECTJ we manually classified 88 bugs where SBFL techniques are suitable. Notably, only 11 bugs of these bugs can be found after examining the 1000 most suspicious lines and on average 250 source code files need to be inspected per bug. Based on these results, the study showcases the limitations of SBFL on a larger program and points out areas for improvement.**

I. INTRODUCTION

Software developers spend a large proportion of their time in finding and removing faults [1]–[4]. Several techniques [5] pursue improving the efficiency of debugging processes. Among them, Spectrum-Based Fault Localization (SBFL) [2] aims at localizing and ranking code elements suspected to cause failing test cases, thus directing the developer towards likely bug locations. The suspiciousness of a code element is determined as a function of the number of successful and failing test cases that executed that code element. Intuitively, high involvement with failing test cases as well as low involvement with successful ones push up suspiciousness. Starting with TARANTULA [2], a variety of SBFL approaches with different suspiciousness ranking metrics have been proposed. Naish et al. [6] compared 33 SBFL approaches based on a simplified single-bug application model and Xie et al. [7] provide a theoretical evaluation. Most approaches in this area [2], [4], [8]–[17] have been evaluated based on analytic effectiveness metrics computed on a set of small benchmarks from the Software-artifact Infrastructure Repository (SIR) [18]. Just recently, an evaluation [19] with medium-sized programs via the Defects4J benchmark [20] has been made available.

However, despite the results reported on small and medium-size benchmarks, SBFL's practical effectiveness is controversial [14], [21]–[23]. As an example, Parnin and Orso [21] provide empirical evidence that SBFL techniques assume idealized behaviors from which real developers often divert.

Consequently, in this paper we evaluate 33 SBFL techniques on a larger-scale application, namely ASPECTJ *[24], finding a poor effectiveness of the state-of-the-art SBFL approaches.*

ASPECTJ(∼500kLoc) has been used by Dallmeier and Zimmermann [25] to extract a benchmark suite of real-world bugs (iBugs) that were semi-automatically mined from the project history and thus provides an ideal foundation for this study. For each bug in this benchmark suite, we manually classified the associated change set of the version control system to identify the real fault location. We executed all the available test cases for each bug and recorded a line hit-spectra to perform SBFL on. The classified faults served as oracle to measure the SBFL performance using different effectiveness metrics.

SBFL techniques are usually evaluated with metrics like proportional Wasted Effort (WE) or Proportion of Bugs Localized (PBL) [8], [10], [12], [13], [17], [26], which are based on the percentage of code that is or needs to be inspected. Since Parnin and Orso [21] already pointed out the inadequacy of these metrics when dealing with larger-scale programs, we use the absolute Wasted Effort and the alternative metric Hit@X [14] that limits the reported suspected elements to a number X and reports the number of bugs that can be found by inspecting these X elements. Lucia et al. [14] argue that for larger projects the likelihood of multiple elements ranked with the same suspiciousness increases. Consequently, for these elements, the inspection ordering becomes vague. Furthermore, Parnin and Orso [21] point out that developers do not follow the list of suspicious elements sequentially. Instead, when inspecting a line of code they also inspect other related lines in the same file or in related classes, usually following element definition chains. Inspired by these two observations, we propose two novel metrics: Area Between Curves (ABC) and Number of Files Inspected (NFI). The former analyzes for the metrics WE and PBL the best and the worst case, and reports the size of the area between these curves. The latter quantifies the number of files that needs to be inspected by developers before finding a real bug.

In summary, the main contributions of this paper are:

- We perform a manual fault classification of real-world bugs for the ASPECTJ project to generate a publicly available benchmark for SBFL approaches.
- We evaluate and compare the effectiveness of 33 SBFL approaches on ASPECTJ, showing the limitations of these approaches on a large program.
- We propose two new metrics for understanding and comparing the effectiveness of SBFL techniques.

II. Background and Related Work

A. Spectrum-based Fault Localization (SBFL)

In software systems, the faulty components that need to be localized can be of any abstraction level, e.g. statements, code lines, blocks, methods, classes, or packages. Current research primarily focuses on the statement level, but there are studies examining the capabilities of SBFL for other abstraction levels [27], [28]. To obtain the involvement of low-level software elements like a code line, various spectra types, such as branch count/hit spectra, path count/hit spectra or data-dependence count/hit spectra [29], can be used. To locate a fault, SBFL assigns a suspiciousness score to each spectra element. This score is calculated by counting the number of passed/failed involvements/non-involvements of each spectra element in the test suite and combining these four numbers using a formula, which is henceforth called *ranking metric* [5]. Adapting Abreu et al. [8], the four numbers are denoted as $\langle n_{np},n_{nf},n_{ip},n_{if}\rangle$ where the first index represents the involvement (i) or non-involvement (n) of the spectra element and the second index represents passing (p) or failing (f) executions.

After a suspiciousness score has been assigned to each spectra element using a ranking metric, the statements can be ranked in descending order by their suspiciousness. Current research assumes that a developer can then find the fault by going through the list element by element, starting with the most suspicious element. Prominent ranking metrics are Tarantula[12] and Ochiai[9]:

$$Tarantula{:=}\frac{\frac{n_{if}}{n_{if}+n_{nf}}}{\frac{n_{if}}{n_{if}+n_{nf}}+\frac{n_{ip}}{n_{ip}+n_{np}}}, Ochiai{:=}\frac{n_{if}}{\sqrt{(n_{if}+n_{nf})(n_{if}+n_{ip})}} \quad (1)$$

For details of the other 31 metrics we refer to Naish et al. [6].

B. General Definitions

This section introduces common definitions [5], [8], [14], [27], [30] required to define performance metrics to compare SBFL metrics. An SBFL metric R (eg. Tarantula or Ochiai) is a function that assigns a suspiciousness score to each spectra element $c_i{\in}C$, which is denoted as $susp_R(c_i)$. C is the *set of all spectra elements*. The *set of faulty spectra elements* (eg. faulty statements, code lines or methods) is a subset of all spectra elements $F{\subseteq}C$.

Prominent Bug: Faulty programs may contain multiple bugs. DiGiuseppe and Jones [30] have shown that multiple faults interfere with each other such that some faults can not be localized using SBFL if others are present at the same time. To overcome this limitation, they propose to iteratively run SBFL: locate the first fault, fix the fault, and then start over again until all bugs are fixed. With this approach, the first bug that is found is the most important bug and is denoted as *prominent bug* $f_{pro}{\in}F$:

$$\forall f_j{\in}F\backslash\{f_{pro}\}{:}susp_R(f_{pro}){\geq}susp_R(f_j) \quad (2)$$

Rank: The effectiveness of SBFL algorithms is evaluated by examining the ranking position of the faulty element in the final ranking. However, the ranking position may not be deterministic, as it can occur that multiple ranked elements share the exact same suspiciousness. As SBFL has no additional information on how to rank draws, all elements with the same suspiciousness are randomly ordered. If there are multiple elements with the same suspiciousness as the faulty element, the final ranking thus has a best case and a worst case. In the best case, the faulty element is the first element of all elements with the same suspiciousness:

$$best\_rank_R(c_j){=}|\{c_i{\in}C|sus_R(c_i){>}sus_R(c_j)\}|{+}1 \quad (3)$$

In the worst case, the faulty element is the last element of all elements with the same suspiciousness:

$$worst\_rank_R(c_j){=}|\{c_i{\in}C|sus_R(c_i){\geq}sus_R(c_j)\}|{+}1 \quad (4)$$

As the random order follows a uniform distribution, the average case is defined by:

$$avg\_rank_R(c_j){=}\tfrac{1}{2}(best\_rank_R(c_j){+}worst\_rank_R(c_j)) \quad (5)$$

With these ranking functions, it is possible to create performance metrics for SBFL techniques.

C. Common Effectiveness Metrics for SBFL

Traditionally, the research community commonly uses two metrics to assess the effectiveness of SBFL, namely the *Wasted Effort* and the *Proportion of Bugs Localized* metric. Recently, new metrics such as the *Hit@X* metric have been proposed. This section defines and explains these metrics.

Wasted Effort (WE): Current research practice assumes that developers have a perfect fault understanding and can identify the fault as soon as they reach the fault location while traversing the ranking list. With this assumption, every developer has to inspect all the elements that are ranked higher than the faulty element. The number of inspected non-faulty elements is thus defined as the wasted effort for the developer. Current research commonly puts the wasted effort in relation to the total number of ranked elements and as such the wasted effort is a percentage defined as:

$$min\_we_R(c_j){=}\tfrac{best\_rank_R(c_j){-}1}{|C|} \quad (6)$$

$$max\_we_R(c_j){=}\tfrac{worst\_rank_R(c_j){-}1}{|C|} \quad (7)$$

As this metric is defined for faulty spectra elements and not the ranking metric itself, the results of this metric need to be aggregated by a function a, for example by taking the average, to compute a score for a specific ranking metric:

$$min\_we\_aggr(R){=}a(\{min\_we_R(f_j)|f_j{\in}F\}) \quad (8)$$

$$max\_we\_aggr(R){=}a(\{max\_we_R(f_j)|f_j{\in}F\}) \quad (9)$$

Using the aggregated score, different ranking metrics can be compared against each other.

Proportion of Bugs Localized (PBL): Another commonly used effectiveness metric is the proportion of bugs localized when examining a certain percentage of code. This metric is defined for a ranking metric and directly produces a score that can be compared to other ranking metrics:

$$min\_pbl_p(R){=}\tfrac{|\{f{\in}F|min\_we_R(f){\leq}p\}|}{|F|} \quad (10)$$

$$max\_pbl_p(R){=}\tfrac{|\{f{\in}F|max\_we_R(f){\leq}p\}|}{|F|} \quad (11)$$

where $p{\in}[0,1]$ is the percentage of code inspected.

Hit@X: Lucia et al. [14] have introduced a new fault localization effectiveness metric called "Hit@10". The key idea is to count how many bugs can be found when investigating a fixed amount of ranked elements. The metric addresses the issues Parnin and Orso [21] have raised, as developers only investigate an absolute number of ranked elements before giving up and using alternate debugging methods. The authors have chosen 10 as threshold for the metric. For this study, the metric is generalized as follows to have a varying threshold of X ranked elements:

$$min\_hit\_count_X(R) = |\{f \in F | best\_rank_R(f) \leq X\}| \quad (12)$$

$$max\_hit\_count_X(R) = |\{f \in F | worst\_rank_R(f) \leq X\}| \quad (13)$$

$X \in \mathbb{N}^+$ represents the number of elements inspected.

III. Study Design

A. Previous Findings and Research Questions

Previous experiments [9], [11], [14] with smaller programs and an empirical study of ranking-based automatic debugging techniques with 68 developers [21] indicate or predict the following problems for the practicability of spectrum based fault localization techniques on large-scale software systems, as detailed below:

- SBFL techniques reveal faulty statements only after inspecting a large number of lines or code elements [21].
- SBFL techniques often assign the same suspiciousness scores for multiple lines or code elements [14].
- Users of SBFL techniques do not inspect in the order provided by the ranked list [21].
- SBFL techniques require a large number of failing test cases to accurately reveal a fault [9], [11].

The goal of this paper is to investigate if these problems really exist for a complex software system.

SBFL techniques report faulty program statements only after inspecting a large number of lines or code elements. Based on the current effectiveness of SBFL techniques [6], [8], [30], as identified by percentage-based metrics, e.g., the Wasted Effort (WE) metric (see Equation 6 and 7) or the Proportion of Bugs Localized (PBL) metric (see Equation 10 and 11), developers need to inspect several thousand lines for large-scale software systems with hundred thousand lines of code and more. This is not feasible in practice and is considered a major drawback for SBFL techniques [21].

RQ$_1$ *What is the absolute SBFL effectiveness?* (Section V-A)

SBFL techniques often assign the same suspiciousness scores for multiple lines or code elements. SBFL algorithms use four numbers $\langle n_{np}, n_{nf}, n_{ip}, n_{if} \rangle$ to compute the suspiciousness. The sum of them is equal to the number of available execution traces (test cases). If only a few test cases are available, the number of possible different inputs to SBFL algorithms drastically shrinks and the amount of different suspiciousness scores is limited. Consequently, multiple program elements will share the same suspiciousness.

The real drawback of common suspiciousness scores arises when ordering the program elements for the developer to inspect. If multiple program elements share the same suspiciousness, those elements cannot be distinguished and the actual order can only be chosen randomly. As an example, Figure 1 shows that even for relatively small programs like `flex`, `gzip`, `grep` and `sed`, there is an observable difference in the ranking (which may accentuate for larger systems).

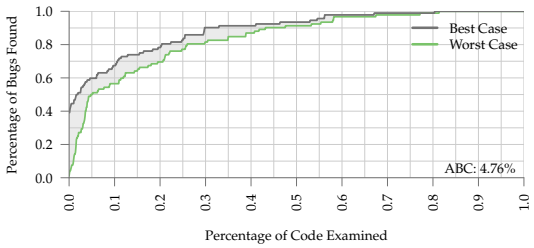


FIG. 1: RESULTS FOR THE PBL METRIC FOR TARANTULA FOR THE SIR PROGRAMS `FLEX`, `GZIP`, `GREP`, AND `SED`.

RQ$_2$ *What is the uncertainty in the assigned suspiciousness scores?* (Section V-B)

Users of SBFL techniques do not inspect in the order provided by the ranked list. Parnin and Orso [21] have shown in their study that developers do not linearly follow the ranking produced by SBFL. Instead, they use the statements ranked high by SBFL as starting points for their investigation and then *search* the surrounding method, class, or file for the actual fault location. This observation indicates that it is more important to point developers to good starting points using SBFL than to improve the ranking of the fault locations itself.

RQ$_3$ *What is the number of files inspected when following SBFL techniques?* (Section V-C)

SBFL techniques require a large number of failing test cases to accurately reveal a fault. The results in Abreu et al. [9] show that on the SIR benchmark [18] even a small number of failing test cases ($n_{nf}+n_{if}$) provide a reasonable fault localization performance. However, the study also recommends that more failing test cases are always better.

RQ$_4$ *What is the relation between the number of failing test cases and SBFL's accuracy?* (Section V-D)

B. Study Subject

To answer our research questions, we have chosen ASPECTJ [24] as a case study. ASPECTJ is an extension to Java that enables developers to use an aspect-oriented programming style. The selection of ASPECTJ is based on the size (∼500kLoc) and the long available development history. Furthermore, Dallmeier and Zimmermann [25] have used ASPECTJ in their iBugs repository and argue that it covers a realistic composition of different, commonly occurring bugs.

C. Study Protocol

To study the suitability of the different SBFL techniques with respect to the AspectJ bugs reported in the iBugs repository, we use two phases. In the first phase, we analyze the bugs manually and identify the faulty line(s) of code from the

change sets and the bug description provided in iBugs. In the second phase, we use the results of the first phase as the data for a descriptive evaluation and a comparison of all 33 used SBFL techniques summarized by Naish et al. [6]. The goal of this descriptive evaluation is to answer our research questions (**RQ**$_1$-**RQ**$_4$). Specifically, we will use the absolute wasted effort metric and the Hit@X metric (Equations 12 and 13) to answer **RQ**$_1$. To answer **RQ**$_2$ and **RQ**$_3$, no suitable metric is available. Consequently, we needed to derive new metrics to cover these two research questions. These two metrics, which are detailed in the remainder of this section, are the *Area Between Curves* (ABC) metric (Equation 14) for research questions **RQ**$_2$ and the *Number of Files Investigated* (NFI) metric (Equations 16 and 17) research questions **RQ**$_3$. For **RQ**$_4$, we will investigate the correlation between the number of failing test cases and minimum wasted effort metric.

1) Area Between Curves (ABC): The past metrics definitions always define a metric for the best and the worst case (and implicitly, due to the uniform distribution, also an average case). However, if the best and worst case of any metric diverge, there have to be ranked elements with the same suspiciousness in the ranking implying a random order to parts of the ranking. To measure the divergence of a given best/worst case metric, the area between the best- and worst-case curves can be leveraged. As $M_{worst} \leq M_{best}$ holds in all points for a metric M, the *area between the curves* is computed by:

$$ABC(M) = AUC(M_{best}) - AUC(M_{worst}) \quad (14)$$

$AUC(M)$ ist the *area under the curve* for a metric M:

$$AUC(M) = \frac{1}{n} \cdot \left(\frac{y_0}{2} + \sum_{i=1}^{n-1} y_i + \frac{y_n}{2} \right) \quad (15)$$

To analyze SBFL techniques with the ABC metric, the PBL metric and the Hit@X metric can be used. Both have a varying parameter that can be used to turn the metric into a discrete cumulative function consisting of a set of equidistant points (x_i, y_i), $i \in [0,n]$, $n \in \mathbb{N}^+$. For a fixed number of support points n, the set of support points for the PBL metric is defined as

$$x_i = \frac{i}{n} \qquad y_i = (min|max)\_pbl_{x_i}(R)$$

and for the Hit@X metric as

$$x_i = \frac{i}{n} \cdot |F| \qquad y_i = (min|max)\_hit_{x_i}(R).$$

The ABC metric reflects the decidedness of a ranking metric. If the randomness in the ranking is low, ABC is close to zero and the decidedness is high. On the other hand, if the randomness in the ranking is high, ABC is high and the decidedness is low. If not otherwise noted, throughout this study the ABC value is presented as percentage of the maximum area that can be achieved by a completely diverging best and worst case. This makes the metric easily comparable if the x-axis or y-axis scale is different.

2) Number of Files Investigated (NFI): To assess the effectiveness of pointing a developer to the right places using SBFL when he/she follows the list of ranked statements in a linear order, we define the *number of files investigated* metric. This metric determines the number of files that need to be investigated before the bug is found.

TABLE I: THE NUMBER OF BUGS USED IN THIS STUDY, ASSOCIATED WITH DIFFERENT COMPONENTS OF ASPECTJ.

Product	ASPECTJ							AJDT		Sum
Component	AJBrowser	AJDoc	Ant	Compiler	IDE	Library	LTWeaving	Core	UI	
All bugs	9	21	33	1287	106	10	78	409	477	2430
Bugs in iBugs	1	8	4	231	19	3	19	19	46	350
Applicable bugs	0	1	1	72	7	0	3	4	0	88
Involved bugs	0	0	1	50	2	0	1	3	0	57

The metric can be defined for the best-case and the worst-case as follows, where $file(c_k)$ returns the file name of the program element c_k:

$$min\_nfi_R(c_j) = |\{file(c_k) | best\_rank_R(c_k) \leq best\_rank_R(c_j)\}| \quad (16)$$

$$max\_nfi_R(c_j) = |\{file(c_k) | worst\_rank_R(c_k) \leq worst\_rank_R(c_j)\}| \quad (17)$$

IV. DATA COLLECTION & PREPARATION: THE ASPECTJ CASE STUDY

We mined all 350 bugs from the iBugs repository [25] and analyzed the specific source code changes, the compiled pre-fix/post-fix versions, and available test cases. The 350 bugs in the ASPECTJ iBugs benchmark suite range from the ASPECTJ bug ID 28919 to ID 173602 and were reported between December 30, 2002 and February 9, 2007, spanning four years of development history. In the examined development timespan, the contributors have produced a total of 7677 commits for a system spanning ~200,000 to ~500,000 lines of code (data taken from Openhub [31]).

A. Data Collection

The iBugs benchmark [25] includes for each bug the change set extracted from the version control system. In most cases however, the change set includes a lot of changed lines and files that did not fix the actual bug. To evaluate the quality of the rankings produced by the various SBFL algorithms in this study, the lines that really contain the fault need to be extracted from the change set. We assumed that the given change set from the iBugs benchmark suite includes the real fault and manually classified all lines in the change set either as *buggy* or *healthy*. In addition to that, a level of confidence (low, normal, or high) including an explanatory comment for all lines classified as *buggy* was stored.

In total, all 350 buggy versions from the benchmark suite were classified and only 88 were applicable for SBFL. The 262 removed buggy versions were not applicable for SBFL for the following reasons: 86 buggy versions were classified as *enhancement* and not as a bug. Section IV-B will expound upon the difference between bugs and enhancements. An additional 111 bugs did not contain a single line in the change set that was classified as fault location, because some of the change sets did not include changes to Java source code at all while others did change Java source code that does not appear in coverage reports (e.g., refactoring names and changing imports). The remaining 65 bugs either had only

faulty lines classified with low confidence or did not produce any execution traces due to compile time or runtime errors. From the set of 88 buggy versions, only fault locations of 57 versions were executed by at least one test case. In this paper, we also use the term *involved bugs* for these 57 versions.

Table I shows the number of bugs and how the set of bugs was reduced through the various stages of data preparation. All bugs in the iBugs benchmark suite were classified for this study, from which $\sim$25% were applicable for SBFL and $\sim$15% were actually involved in at least one test case.

B. Fault Classification

Dallmeier and Zimmermann [25] provide the change set between pre- and post-fix versions in the diffformat between the two corresponding revisions in the version control system. As the changed lines the developers commit mostly exceed the real fault location [32], [33], the change sets had to be manually classified in order to be able to use the data as a benchmark for fault localization. During the classification, several bugs were considered to be difficult to localize using SBFL. This section summarizes the evidence found.

a) Misclassification of Bug Reports: After a careful inspection of the bug description and the change set itself, 86 bugs were considered not to be a real bug. Herzig et al. [34] propose a set of rules which are used to distinguish bugs from non-bugs. Those rules were applied during the classification process to judge whether a reported bug is a real bug or not. In the positive case, a bug report *increases* the likelihood of the bug report being a real bug, among others, if:

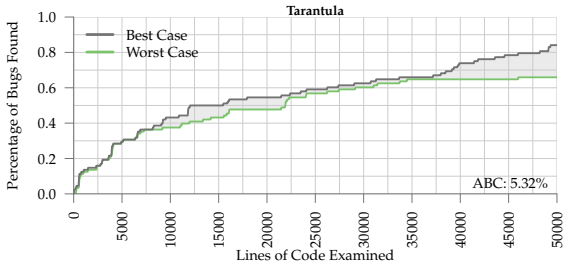
- It reports a `NullPointerException` (most common exception type) or other types of exceptions.
- A minimal failing example was provided that leads to small semantic code changes fixing the issue.

For the negative case, we identified the following rules that *decrease* the likelihood of a bug report being a real bug, for example, if:

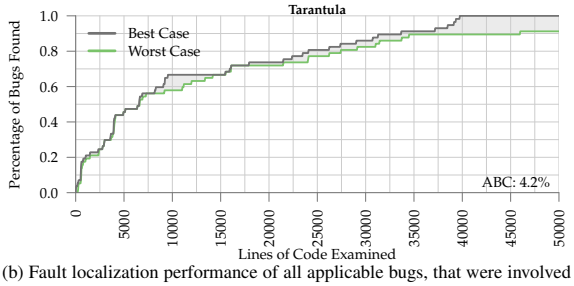
- The phrases "it would be great if", "improve" or "enhance", or similar phrases appeared in the bug description. No automatic classification was performed, but the presence of these words have a high correlation with the bug report not being a real bug, but rather an enhancement.
- The report contains a request to add new features.
- Only strings (exception strings, log strings, ...) were changed.

The classification was performed in a restrictive manner, such that only bugs with a high likelihood of being a bug were actually classified as a bug. The goal was to minimize the type I error for the classification.

b) Bugs that are Difficult to Detect Using SBFL: Besides the misclassification of bugs, there are some bugs that are difficult to detect with SBFL, as they are generally difficult to reproduce. Several bugs occurred due to concurrency issues or environment issues (a limited number of reproducible patterns have been successfully analyzed in [35]). Environment issues include for example hardware constraints



(a) Fault localization performance of all bugs using Tarantula .



(b) Fault localization performance of all applicable bugs, that were involved in at least one test case, using Tarantula .

FIG. 2: ABSOLUTE FAULT LOCALIZATION PERFORMANCE.

that lead to `OutOfMemoryError`'s that were resolved by adding appropriate try-catch clauses. Another fix resolved a concurrency issue by adding a `synchronized` modifier to a method. Both example cases may be difficult to reproduce in a test suite within reasonable budget constraints and are thus difficult to detect using SBFL as they may not lead to deterministically failing tests in the test suite.

> From the 350 Bugs present in the iBugs repository only 88 were found to be applicable for SBFL. A large number of bugs (86) were actually no bugs at all. From the 88 applicable SBFL bugs only 57 bugs were executed by at least one failing test case. Consequently, for a large scale software system, only a low proportion of bugs may be identifiable with SBFL techniques.

V. EXPERIMENTAL RESULTS

A. RQ_1: What is the absolute SBFL performance?

To understand the effect of the program size, we will present the results of the Absolute Wasted Effort and Hit@X Metric in the following. We focus on the Tarantula and Ochiai ranking metrics as representative and well-known ranking metrics.

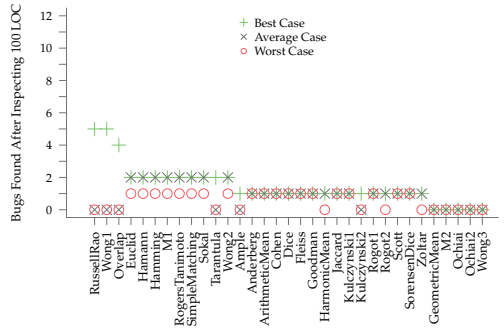
1) Absolute Wasted Effort: Figure 2a shows the percentage of bugs found using the Tarantula ranking metric after examining 50,000 lines of code. When trying to locate 50% of all fault locations, in the worst case more than 20,000 lines of code need to be inspected. When only trying to locate 20% of all fault locations, a developer is still required to inspect around 3,000 lines of code using the Tarantula technique. These two numbers indicate that the state of the art of SBFL needs to be improved to effectively support developers.

In Figure 2b, only fault locations that are involved at least once in a passing or failing test case are examined. Under this condition, the performance slightly improves compared to Figure 2a. However, SBFL users usually cannot guarantee

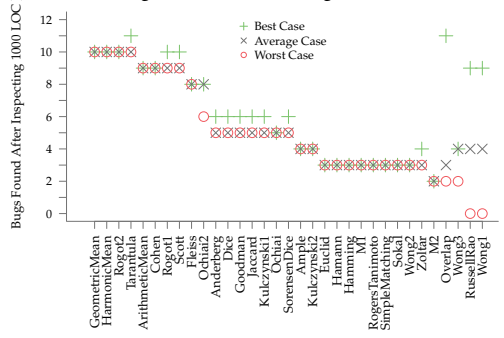
that the test suite will cover the fault localization and especially if the fault location is part of a newly written code segment. Consequently, techniques that improve the quality of the test suite need to be combined with SBFL techniques, c.f., [26], [36]–[38]. Specifically interesting are regression test case generation approaches [39], since bugs in newly written code are often not covered by existing test cases.

> To locate more than 20% or 50% or all bugs, a developer is required to inspect more than 7,000 and 20,000 ranked elements, respectively. However, fault locations that are involved in at least one failing or passing test case are ranked significantly higher.

2) Hit@X: Figure 3a shows the Hit@100 metric (Equation 12) for all examined SBFL ranking metrics. The vertical axis represents the number of bugs found after examining the first 100 ranked elements produced by each ranking metric. The plot marks the best-, average-, and worst-case for each ranking metric. For the Hit@100 metric, the average and worst-case are all less than or equal to two, which means that all ranking metrics find less than 4% of all applicable, involved bugs within the first 100 ranked lines of code in the average or worst-case. In the best case, a third of the ranking metrics is able to locate at least two bugs. The data suggests that investigating only the first 100 ranked elements does not help the developer at all. Nevertheless, interpreting and investigating 100 ranked elements is already very much work for a developer, and is not really realistic [21]. Furthermore, Lucia et al. [14] state that inspecting 10 ranked elements would be a more realistic number. In the best-case, some



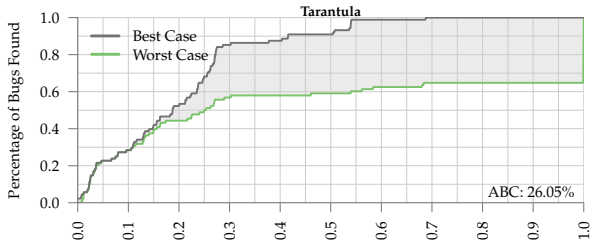
(a) Number of bugs found after examining 100 ranked lines.



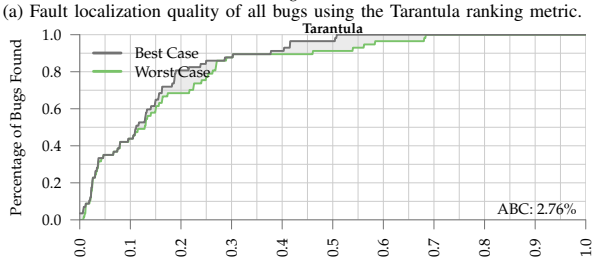
(b) Number of bugs found after examining 1000 ranked lines.

FIG. 3: HIT@100/1000 METRIC FOR ALL SBFL TECHNIQUES.

SBFL techniques find five of the 88 prominent fault locations.



(a) Fault localization quality of all bugs using the Tarantula ranking metric.



(b) Fault localization quality of all bugs that were involved in at least one execution trace using the Tarantula ranking metric.

FIG. 4: TARANTULA FAULT LOCALIZATION DECIDEDNESS.

However, on average the techniques find only one to two of these faults after examining 100 ranked elements.

Figure 3b shows the Hit@1000 metric for all SBFL ranking metrics/techniques. When examining the first 1000 ranked elements, nearly all ranking metrics guarantee to find at least two bugs. The median of the number of found bugs across all ranking metrics is five. Nevertheless, 1000 ranked elements are too many elements for a developer to examine in practice while in the best-case (11 out of 88 bugs), it is only possible to find ∼12,5% of all applicable bugs. In addition to that, a deeper analysis of all identified bugs reveals that they are involved in at least one failing test case, which implies that no bug that is not involved can be found within the first 1000 ranked elements. This confirms and strengthens the results from Section V-A1.

> Applying the Hit@100 and Hit@1000 metric, at most 2 and 5 of the 88 bugs were found in the average case by all ranking metrics, respectively. In the best case, only up to 5 and 11 bugs are identifiable when developers examine the top 100 and 1000 ranked elements, respectively.

B. RQ_2: *What is the uncertainty in the assigned suspiciousness scores?*

The examined versions of ASPECTJ contain a large number of of executable and coverage-producing lines of code (cp. Section IV). For each version, between 990 and 1,999 execution traces were produced. Under the given circumstances, most suspiciousness scores in the ranking are shared by multiple lines. As the order of multiple lines with the same suspiciousness can only be randomly chosen, the Area Between Curves (ABC) metric (see Section III-C1) has been defined as a repeatable and meaningful metric. In all plots, the area used to calculate the ABC metric between the two curves is highlighted. The smaller the ABC metric is, the less random

a ranking is and, as such, the results of the fault localization algorithm are more reliable for a developer debugging a program. A value of 100% represents a completely random ranking (low decidedness), whereas a value close to 0% (high decidedness) represents a deterministic ranking.

1) ABC for Involved and Non-Involved Bugs: Figure 4 shows the fault localization quality of Tarantula [2]. Both figures plot the best- and worst-case proportion of prominent bugs localized. Figure 4a includes all prominent bugs in the dataset, whereas Figure 4b only includes all prominent bugs that were involved in at least one test case. In both plots, the horizontal axis represents the percentage of code examined and the vertical axis represents the percentage of bugs localized. The curves reveal the minimum (worst-case) percentage of bugs a developer can find and the maximum (best-case) percentage of bugs a developer can find when examining a certain percentage of the ranked lines.

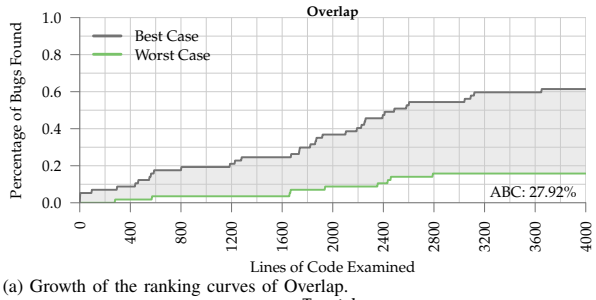
In Figure 4, Tarantula maps a lot of fault locations to the same suspiciousness resulting in an ABC of 26.05%. After examining $\sim 25\%$ of the code, the Tarantula ranking shows a very large divergence between the best- and worst-case ranking and thus becomes worthless from a practical point of view, as all remaining bugs are in random order. However, approximately a third of the evaluated bugs were never involved in either a passing or failing execution trace. As already shown in Section V-A1, under such a condition, it is difficult for spectrum-based fault localization to locate the fault, as the fault location then shares the same suspiciousness with at least all the statements that also were never involved in a single execution trace.

The decidedness improves, as indicated by an ABC metric of 2.76% and the curves in Figure 4b, if the SBFL technique is only locating faults that are involved in at least one execution trace. However, it is important to keep in mind that software engineers do not know if the test suite has actually covered the bug and thus there is a test case that involves the bug.

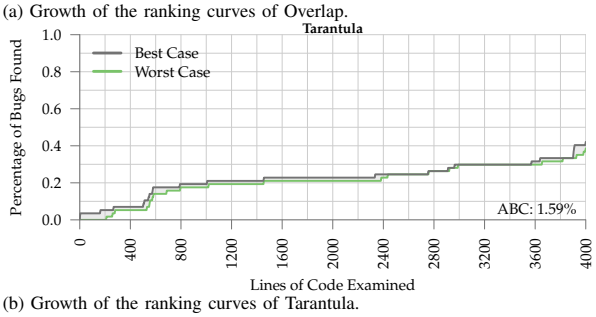
2) Impact of Different Ranking Metrics: SBFL intends to point the developer to the fault location with the first elements in the ranking and thus, algorithms should rank elements in the first positions with high confidence and decidedness. Examining the growth of the best- and worst-case rankings for the first $4{,}000$ lines of code for the Overlap ranking metric in Figure 5a, it is evident that the guidance for a developer is not very clear. Depending on the random ranking, a developer may find up to $\sim 20\%$ of the bugs after examining $1{,}000$ lines of code or less than 5%. Overlap has the highest ABC value of all examined ranking metrics in the first $4{,}000$ lines of code.

Figure 5b shows the ranking curves for the Tarantula ranking metric with an ABC metric of 1.59%, being clearly superior to the results for the Overlap ranking metric with an ABC metric of 27.92%.

All in all, if multiple elements share the same suspiciousness, SBFL reaches its limitations. The examined ranking metrics considerably differ in their ABC value for programs of large size and thus, the ABC value may be used as an indicator



(a) Growth of the ranking curves of Overlap.



(b) Growth of the ranking curves of Tarantula.

FIG. 5: A CLOSER LOOK AT THE FAULT LOCALIZATION DECIDEDNESS IN THE FIRST 4,000 LINES OF EXAMINED CODE.

whether a given ranking metric is suited for fault localization for a given problem domain with a certain size.

> Large programs can introduce randomness in the ranking through a lot of ranked elements having the same suspiciousness. If the fault locations are guaranteed to be involved in at least one execution trace, ranking metrics are much more decided, resulting in lower ABC values. Different ranking metrics may have an impact on the ABC value.

C. RQ_3: What is the number of files inspected when following SBFL techniques?

The Number of Files Investigated metric counts the files a developer looks at until the prominent bug is found. As defined in Section III-C2, it is assumed that developer open each file only once, inspect the complete file, and then skip all program elements in the ranking that where already investigated the file the program element is contained in.

Figure 6 plots the average number of files a developer has to investigate prior to finding the file containing the prominent bug for all buggy versions in which the prominent bug was involved in at least one execution trace. The vertical axis contains an entry for each of the 33 ranking metrics [6]. The horizontal axis represents the average number of files a developer has to investigate and is plotted on a logarithmic scale. The minor ticks mark the arithmetic mean between the next lower and higher major ticks. The logarithmic scale allows to spot the differences for the buggy versions where less than 100 files have to be investigated while still allowing to show the large part of the data where more than 100 files have to be investigated, and thus emphasizes the data that is most important to developers: the performance of the first investigated files. Each line in the plot represents the average number of files that need to be investigated for a specific buggy version of ASPECTJ using the ranking produced by the

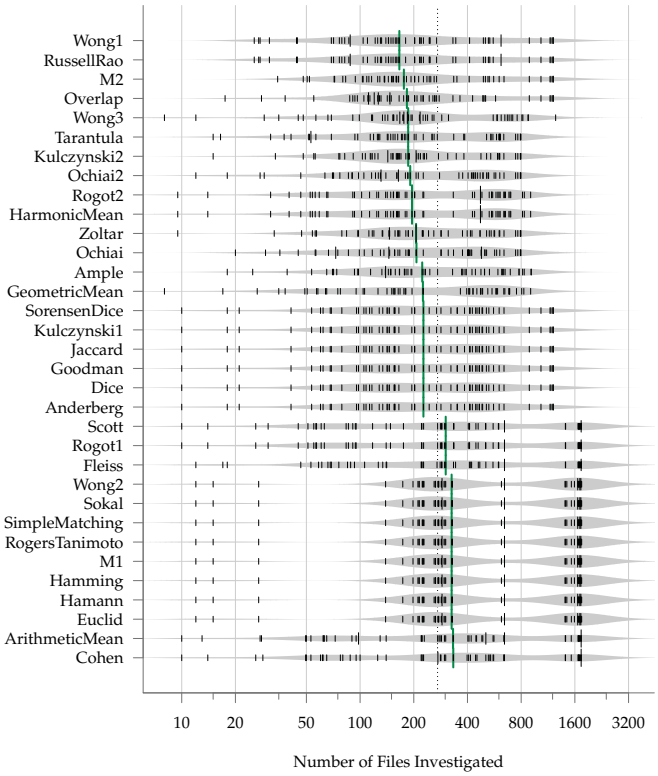


FIG. 6: AVERAGE NUMBER OF FILES INVESTIGATED IN ORDER TO FIND ALL PROMINENT BUGS THAT WERE INVOLVED AT LEAST ONCE IN A COVERAGE TRACE.

according ranking metric. A longer line indicates that multiple prominent bugs can be localized with the exact same average number of files. The thick long line represents the median of the number of investigated files for all buggy versions examined for the according ranking metric. The dotted line in the background is the median of the medians of the ranking metrics. The density of the buggy versions is represented by the shaded area. A larger shaded area means that a larger number of prominent bugs can be found within the effort range. The ranking metrics are sorted by their median in ascending order.

For most ranking metrics a developer has to investigate more than 10 files on average to find the first prominent bug. In order to find 50% of the bugs, a developer has to investigate more than 150 files on average using the best ranking metric, which we consider an unfeasible task in practice. For all SBFL ranking metrics, the bulk contribution to the shaded area is around the median of the respective metric. That means that it is very likely to find a lot of bugs when examining an average number of files that is close to the median of the respective ranking metric. However, the median of the median of all ranking metrics is nearly 250 files (exact measurement: 226.5 files) that need to be investigated.

In summary, the data clearly indicates that a developer has to inspect a large number of files in the fault localization process. This is in our opinion a major draw back of the SBFL techniques and also opens opportunities for improvement of the SBFL approaches.

> The NFI metric indicates that developers using SBFL techniques on the ASPECTJ case study have to inspect on average 250 files per bug.

D. RQ_4: What is the relation between the number of failing test cases and SBFL's accuracy?

Figure 7 shows the relation between the number of failing test cases and the minimum wasted effort in scatter plots for Tarantula and Ochiai. Please note that a lower wasted effort indicates a higher SBFL accuracy. A manual inspection of Figure 7a and 7c reveals that there is no particular relation between the two variables. This can be explained with the same argument as provided in Section V-A that failing test cases that do not involve the target bug provide limited information to localize the bug. An investigation of the theoretical case with only failing test cases that involve the bug (in Figure 7b and 7d) hints at a trend that more involved failing test cases improve the accuracy of SBFL techniques.

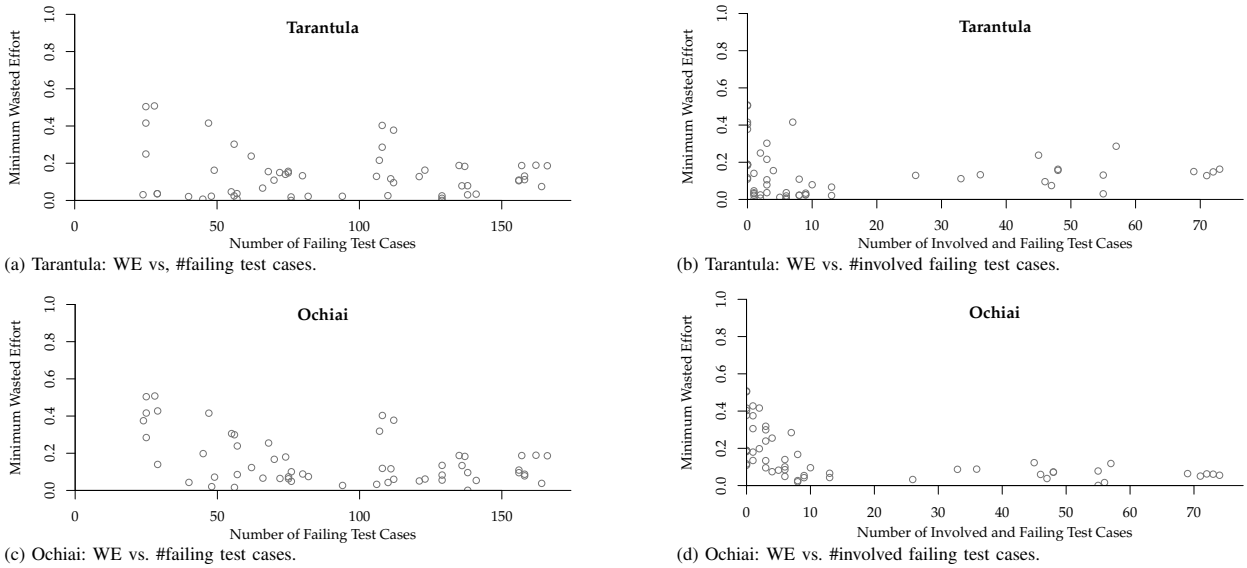
For a formal analysis of the relationship, we analyze the correlation between number of test cases and minimum wasted effort. Based on the data, as given in (Figure 7a-7d), non-parametric correlation coefficients need to be computed. The resulting Spearman correlation coefficients are -0.026 (p-value$>$0.05) and -0.313 (p-value$<$0.05) for Tarantula and Ochiai, respectively. For all 33 SBFL metrics [6], the median correlation coefficient is -0.036 with a variance of 0.098. This indicates that there is no correlation. However, the analysis is rather inconclusive due to the high variance of the correlation coefficients due to the inclusion of basic and weaker SBFL Metrics (cp. [6]) and some high p-values for the smaller correlation coefficients. The second data set considers only involved failing test cases (Figure 7b and 7d) and the correlation coefficients are -0.182 (p-value$>$0.05) and -0.750 (p-value$<$0.05) for Tarantula and Ochiai. Thus, the correlations are higher than for the data that considers all failing test case. This supports the previous observations that test case quality matters, and this gives the community a hint that more high quality test cases which actually execute the bugs improve SBFL's fault localizing capabilities.

> The data suggests but does not confirm that there is no correlation between the number of failing test cases and the wasted effort metric. Furthermore, there is an indication that more involved test cases can improve SBFL's accuracy.

E. Effectiveness Metrics in Comparison

Table II shows various effectiveness metrics for all examined SBFL ranking metrics. All columns but the last column are percentage values rounded to two digits, whereas the last column is the absolute value of the NFI effectiveness metric. The values were computed as defined in Section III. The best value of each effectiveness metric is printed in bold font.

For the $\{min,max\}\_we\_aggr$ metrics, the wasted effort metric of all bugs has been aggregated using the arithmetic



(a) Tarantula: WE vs, #failing test cases.

(b) Tarantula: WE vs. #involved failing test cases.

(c) Ochiai: WE vs. #failing test cases.

(d) Ochiai: WE vs. #involved failing test cases.

FIG. 7: THE RELATION BETWEEN THE NUMBER OF [INVOLVED] FAILING TEST CASES AND SBFL'S ACCURACY.

mean. ABC (Equation 14) refers to the Area Between Two Curves, namely the area between the min_pbl and max_pbl curves. The last column shows the median number of files investigated (Section III-C2) metric. In addition to the results of the various ranking metrics, the last row of the table contains the arithmetic mean of each effectiveness metric for all ranking metrics.

Naish et al. report a mean value of 12.27% for the average of the $pbl_{0.01}$ metric using programs from the SIR and 23 different ranking metrics [6]. In addition to that, a mean value of 46.57% is achieved when investigating 10% of the code ($pbl_{0.1}$) using the same artifacts. In this study, the mean PBL value is slightly lower. For 1% of examined code ($pbl_{0.01}$), it is 5.05%, and for 10% of examined code ($pbl_{0.1}$) it is (35.09%). That means that with the same percentage based effort, there are less bugs found in the ASPECTJ dataset. Lucia et al. report a mean value of 36.01% for the average of the $pbl_{0.1}$ metric using 23 different ranking metrics and projects from the SIR [14]. The mean values indicate that the effectiveness measured by the PBL metric does not differ in an unexpected way when comparing the results of this study with the results of other researchers with smaller programs from the SIR.

For the wasted effort metric, Abreu et al. report a mean value of 9.86% using four ranking metrics [11] and a mean value of 12.5% for four ranking metrics in a different study [8]. Naish et al. report a mean value for the metric of 22.25% with 25 different ranking metrics [6]. All reported mean values originate from a dataset containing only programs from the SIR. The mean value of the average wasted effort metric in this study is 35.01%. This indicates that the WE metric for this study compared to the results of other researchers using smaller programs is higher. However, as it is more difficult to pinpoint a fault location in a larger program it is not unexpected to have a higher wasted effort.

Section V-A examining the Hit@100 and Hit@1000 metric has shown that the effectiveness of SBFL for a project of this size is too unreliable for a developer to adapt SBFL in practice, as no developer will examine a list of ranked program elements longer than 1000 elements, even when embracing skipping and zig-zag-patterns to scan the list. Hence, even though the values of commonly used effectiveness metrics computed in this study fit into the bounds of common research results, the rankings produced by SBFL in this study are unusable for developers in practice. This means that the used effectiveness metrics are not suited to distinguish between useful and impractical rankings. Thus, the data provides evidence that the commonly used effectiveness metrics are not suited to measure the SBFL effectiveness.

All in all, the data supports that the practical validity of the commonly used effectiveness metrics has to be questioned. Additional effectiveness metrics should be developed, improved and thoroughly validated, for example as described by Schneidewind [40].

The commonly used proportion-based effectiveness metrics are within the bounds of the results of other researchers using the SIR. However, the results for the absolute effectiveness metrics (absolute WE, Hit@X and NFI) are worse for the relatively large ASPECTJ code base. Thus, indicating that the SBFL techniques have to be significantly improved to be applicable in practice.

VI. DISCUSSION

A. Threats to Validity

The are two main potential threats to validity for this work. First, SBFL approaches and the proposed metrics (ABC and NFI) have been evaluated on ASPECTJ only. This case has been selected because 1) it has a relatively large scale compared to the common benchmarks in the area and 2)

TABLE II: VARIOUS SBFL EFFECTIVENESS METRICS FOR THE DIFFERENT RANKING METRICS.

Ranking metric	min_we_aggr	max_we_aggr	$min\_pbl_{0.01}$	$max\_pbl_{0.01}$	$min\_pbl_{0.1}$	$max\_pbl_{0.1}$	ABC	$median\_nfi$
Ample	18.39	18.84	5.26	7.02	35.09	35.09	**0.45**	223.00
Anderberg	15.49	30.86	5.26	5.26	45.61	47.37	15.36	226.50
ArithmeticMean	42.01	42.54	**7.02**	7.02	43.86	45.61	0.53	332.00
Cohen	41.45	41.98	5.26	7.02	42.11	43.86	0.52	333.00
Dice	15.49	30.86	5.26	5.26	45.61	47.37	15.36	226.50
Euclid	65.88	66.82	5.26	5.26	12.28	12.28	0.95	326.00
Fleiss	44.59	45.12	5.26	5.26	40.35	40.35	0.53	302.00
GeometricMean	15.14	**15.67**	5.26	5.26	**49.12**	**49.12**	0.54	225.00
Goodman	15.49	30.86	5.26	5.26	45.61	47.37	15.36	226.50
Hamann	65.88	66.82	5.26	5.26	12.28	12.28	0.95	326.00
Hamming	65.88	66.82	5.26	5.26	12.28	12.28	0.95	326.00
HarmonicMean	15.38	15.90	**7.02**	**8.77**	45.61	45.61	0.51	195.50
Jaccard	15.49	30.86	5.26	5.26	45.61	47.37	15.36	226.50
Kulczynski1	15.49	30.86	5.26	5.26	45.61	47.37	15.36	226.50
Kulczynski2	16.20	18.97	3.51	5.26	38.60	38.60	2.77	186.00
M1	65.88	66.82	5.26	5.26	12.28	12.28	0.95	326.00
M2	17.31	32.66	0.00	0.00	36.84	36.84	15.34	176.00
Ochiai	16.07	18.85	1.75	1.75	47.37	47.37	2.78	207.00
Ochiai2	14.96	17.69	3.51	3.51	47.37	47.37	2.72	191.00
Overlap	**10.25**	37.76	1.75	15.79	31.58	**63.16**	27.51	183.00
RogersTanimoto	65.88	66.82	5.26	5.26	12.28	12.28	0.95	326.00
Rogot1	42.55	43.08	3.51	5.26	43.86	43.86	0.54	302.00
Rogot2	15.38	15.90	**7.02**	**8.77**	45.61	45.61	0.51	195.50
RussellRao	16.53	33.62	1.75	**17.54**	35.09	36.84	17.07	**166.00**
Scott	42.55	43.08	3.51	5.26	43.86	43.86	0.54	302.00
SimpleMatching	65.88	66.82	5.26	5.26	12.28	12.28	0.95	326.00
Sokal	65.88	66.82	5.26	5.26	12.28	12.28	0.95	326.00
SorensenDice	15.49	30.86	5.26	5.26	45.61	47.37	15.36	226.50
Tarantula	13.77	16.54	3.51	7.02	43.86	43.86	2.76	186.00
Wong1	16.53	33.62	1.75	**17.54**	35.09	36.84	17.07	**166.00**
Wong2	65.88	66.82	5.26	5.26	12.28	12.28	0.95	326.00
Wong3	20.90	23.84	5.26	7.02	33.33	33.33	2.95	186.00
Zoltar	16.14	18.90	1.75	3.51	40.35	40.35	2.77	206.00
Mean value	32.0	38.01	4.46	6.43	34.87	36.36	6.01	249.39

its code and bug reports are publicly available [25] through a versioning system. However, there is a threat to external validity that the obtained results cannot be generalized for other large scale software systems. To reduce the threat to external validity of our findings, we have also investigated buggy versions of three large programs (i.e., Rhino, Lucene, Ant)[14]. Specifically, we analyze buggy versions where the bugs span up to 5 lines [14]. Our analysis on these buggy versions also shows similar results which can be found at [41].

A second potential threat to validity lies in the manual classification of the bug locations. In the classification, we stated for each faulty line also a confidence level (low, medium, high). As a conservative choice, for this research, we considered only bugs classified with high confidence. For each non-trivial bug we classified, we documented an argument supporting that classification. It is possible that some bug has been misclassified. For this reason, we made the commented code versions for this experiment available at [41], including our classification and the relative comments to be checked for future investigations.

B. Recommendations

We recommend future SBFL studies to evaluate their proposed approaches beyond smaller programs in SIR. Larger programs can serve as harder yardsticks for SBFL techniques. An SBFL technique that can work well for larger programs is more likely to work well for smaller programs, but it is not likely the other way round.

We also recommend the inclusion of larger programs in SIR. There is a need for a community-wide effort to create a benchmark containing many larger programs of various kinds (compilers, IDEs, web applications, etc.) to fully assess the effectiveness of SBFL and other techniques. As demonstrated in this work, the creation of such benchmark requires much manual effort involving the exclusion of feature (enhancement) requests, locating faulty program elements from fixes, and filtering bugs unsuitable for SBFL.

Our findings also highlight that test suite quality is an important factor that determines the effectiveness of SBFL techniques. A number of past studies have proposed automated test case generation techniques to enrich test suites to improve SBFL effectiveness [26], [36], [37]. However, such studies assume the availability of test oracles, which unfortunately are unavailable for many programs and bugs. Hence, there is a need for more effective and practical approaches that can either enhance SBFL to work well even with poor quality test suites, look at other debugging hints beyond program spectra to locate bugs, or generate test oracles for newly generated test cases. We encourage future studies to investigate these research opportunities.

VII. CONCLUSIONS & LESSONS LEARNED

Debugging large scale programs might be dramatically time consuming [5]. Spectrum-Based Fault Localization is an automated fault localization technique aiming at localizing and ranking a set of suspicious elements likely to be the cause of a failing test case. Several SBFL approaches have been evaluated only on small and medium-size benchmarks and under the unrealistic assumption of a developer sequentially going through the ranked list of suspects [21].

In this paper, we evaluated 33 state-of-the-art SBFL approaches for the debugging of a larger scale project, ASPECTJ. We take the source code versioning repository and the bug reports for ASPECTJ from the publicly available iBugs collection [25]. From an initial set of 350 faulty version from the iBugs repository of ASPECTJ, we manually classified 88 bugs where SBFL techniques can be applied. To compare the different SBFL approaches, we used both a set of established metrics and two additional ones we defined, which try to better approximate more realistic assumptions about developers' behavior. Overall, at most 11 bugs can be found with any of the investigated SBFL techniques after examining the 1000 top ranking suspicious lines, requiring on average an inspection of about 250 files to discover any bug. These results extend earlier studies [8], [19] and show a generally

poor performance of current SBFL approaches on this case study. We summarize the main lessons learned as:

Lesson 1. A large proportion of bugs cannot be directly detected with the studied SBFL techniques, e.g., concurrency bugs or environment related bugs like memory leaks.

Lesson 2. A large proportion of bugs were not executed by a single failing test case (n_{if}=0). Since the involvement in failing test case significantly affects localizability of a bug with SBFL techniques, this limits the effectiveness of SBFL.

Lesson 3. The study shows that for large programs, the number of lines and files that need to be inspected based on the recommendation of SBFL techniques is high, but can be reduced by high quality test cases that execute the bug.

Lesson 4. The study shows that multiple elements share the same SBFL suspiciousness scores, thus introducing randomness into the ranking order.

Lesson 5. There is weak evidence that the number of failing test cases has no correlation with the SBFL's accuracy. It is more important to have high quality test cases that improve the chance of executing the bug.

Lesson 6. The study shows that results that have been obtained by using small programs are hard to replicate/generalize on the ASPECTJ case study.

REFERENCES

[1] H. Cleve and A. Zeller, "Locating causes of program failures," in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05, ACM, 2005, pp. 342–351.

[2] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of the 24th International Conference on Software Engineering*, ser. ICSE '02, ACM, 2002, pp. 467–477.

[3] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05, ACM, 2005, pp. 15–26.

[4] C. Liu, L. Fei, X. Yan, J. Han, and S. Midkiff, "Statistical debugging: A hypothesis testing-based approach," *IEEE Transactions on Software Engineering*, 32, no., pp. 831–848, 2006.

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

[6] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Trans. Softw. Eng. Methodol.*, 20, no., pp. 1–32, Aug. 2011.

[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.*, 22, no., pp. 1–40, Oct. 2013.

[8] R. Abreu, P. Zoeteweij, and A. J.C. v. Gemund, "An evaluation of similarity coefficients for software fault localization," in *12th IEEE Pacific Rim Int. Symposium on Dependable Computing (PRDC 2006)*, IEEE Computer Society, 2006, pp. 39–46.

[9] ——, "On the accuracy of spectrum-based fault localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques*, ser. MUTATION, 2007, pp. 89–98.

[10] L. Gong, D. Lo, L. Jiang, and H. Zhang, "Diversity maximization speedup for fault localization," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '12, 2012, pp. 30–39.

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

[12] J. A. Jones, J. F. Bowring, and M. J. Harrold, "Debugging in parallel," in *Proceedings of the International Symposium on Software Testing and Analysis*, ser. ISSTA '07, ACM, 2007, pp. 16–26.

[13] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "Sober: Statistical model-based bug localization," in *Proceedings of 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. ESEC/FSE '05, ACM, 2005, pp. 286–295.

[14] Lucia, D. Lo, and X. Xia, "Fusion fault localizers," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '14, ACM, 2014, pp. 127–138.

[15] X. Xia, L. Gong, T. B. Le, D. Lo, L. Jiang, and H. Zhang, "Diversity maximization speedup for localizing faults in single-fault and multi-fault programs," *Autom. Softw. Eng.*, 23, no., pp. 43–75, 2016.

[16] W. E. Wong, Y. Qi, L. Zhao, and K.-Y. Cai, "Effective fault localization using code coverage," in *Proceedings of the 31st Annual International Computer Software and Applications Conference*, ser. COMPSAC '07, IEEE Computer Society, 2007, pp. 449–456.

[17] L. Zhao, Z. Zhang, L. Wang, and X. Yin, "PAFL: fault localization via noise reduction on coverage vector," in *Proceedings of the 23rd International Conference on Software Engineering & Knowledge Engineering*, ser. SEKE '11, 2011, pp. 203–206.

[18] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques:

An infrastructure and its potential impact," *Empir. Software Engineering*, 10, no., pp. 405–435, 2005.

[19] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization techniques," in *Proceedings of the ACM/IEEE International Conference on Software Engineering*, ser. ICSE '17, 2017.

[20] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *International Symposium on Software Testing and Analysis, ISSTA 2014*, ACM, 2014, pp. 437–440.

[21] 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, ACM, 2011, pp. 199–209.

[22] T. B. Le and D. Lo, "Will fault localization work for these failures? an automated approach to predict effectiveness of fault localization tools," in *Proceedings of the 29th IEEE Int. Conference on Software Maintenance*, ser. ICSM, 2013, pp. 310–319.

[23] 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 '13, ACM, 2013, pp. 314–324.

[24] *AspectJ language extension*, http://www.eclipse.org/aspectj, Eclipse Foundation.

[25] V. Dallmeier and T. Zimmermann, "Extraction of bug localization benchmarks from history," in *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '07, ACM, 2007, pp. 433–436.

[26] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Practical fault localization for dynamic web applications," in *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10, ACM, 2010, pp. 265–274.

[27] R. Abreu, P. Zoeteweij, and A. J.C. v. Gemund, "A practical evaluation of spectrum-based fault localization," *J. Syst. Softw.*, 82, no., pp. 1780–1792, Nov. 2009.

[28] P. Casanova, B. Schmerl, D. Garlan, and R. Abreu, "Architecture-based run-time fault diagnosis," in *Software Architecture*, ser. LNCS, vol. 6903, Springer, 2011, pp. 261–277.

[29] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi, "An empirical investigation of the relationship between spectra differences and regression faults," *Software Testing, Verification and Reliability*, 10, no., pp. 171–194, 2000.

[30] N. DiGiuseppe and J. Jones, "Fault density, fault types, and spectra-based fault localization," *Empirical Software Engineering*, no., pp. 1–40, 2014.

[31] Openhub, *The aspectj open source project on open hub : Languages page*, https://www.openhub.net/p/freshmeat_aspectj/analyses/latest/languages_summary, [Online; accessed 29-August-2015], 2015.

[32] K. Herzig and A. Zeller, "The impact of tangled code changes," in *Mining Software Repos. (MSR), 2013 10th IEEE Working Conference on*, 2013.

[33] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11, ACM, 2011, pp. 351–360.

[34] K. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: How misclassification impacts bug prediction," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13, IEEE Press, 2013, pp. 392–401.

[35] S. Park, R. W. Vuduc, and M. J. Harrold, "Falcon: Fault localization in concurrent programs," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, ser. ICSE '10, ACM, 2010, pp. 245–254.

[36] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Directed test generation for effective fault localization," in *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ser. ISSTA '10, ACM, 2010, pp. 49–60.

[37] J. Campos, R. Abreu, G. Fraser, and M. d'Amorim, "Entropy-based test generation for improved fault localization," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, 2013, pp. 257–267.

[38] J. Rößler, G. Fraser, A. Zeller, and A. Orso, "Isolating failure causes through test case generation," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ser. ISSTA, ACM, 2012, pp. 309–319.

[39] W. Jin, A. Orso, and T. Xie, "Automated behavioral regression testing," in *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Paris, France, April 7-9, 2010*, IEEE Computer Society, 2010, pp. 137–146.

[40] N. F. Schneidewind, "Methodology for validating software metrics," *IEEE Trans. Softw. Eng.*, 18, no., pp. 410–422, May 1992.

[41] 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 - Additional Material*, www.informatik.hu-berlin.de/de/forschung/gebiete/se/research/material/QRS2017.