Received 26 April 2016; | Revised 6 June 2016; | Accepted 6 June 2016

DOI: xxx/xxxx

ARTICLE TYPE

An Evaluation of Pure Spectrum-Based Fault Localization Techniques for Large-Scale Software Systems

Simon Heiden*[1] | Lars Grunske[1] | Timo Kehrer[1] | Fabian Keller[2] | Andre van Hoorn[2] | Antonio Filieri[3] | David Lo[4]

[1]Institute of Computer Science, Software Engineering group, Humboldt-Universität zu Berlin, Unter den Linden 6, Berlin, Germany

[2]Institute of Software Technology, Reliable Software Systems Group, Universität Stuttgart, Universitätsstrasse 38 , Stuttgart, Germany

[3]Department of Computing, Imperial College London, 180 Queen's Gate, London, UK

[4]School of Information Systems, Singapore Management University, 80 Stamford Road, Singapore, Singapore

Correspondence
*Corresponding author,
Humboldt-Universität zu Berlin,Unter den Linden 6, Berlin, Germany Email: heiden@informatik.hu-berlin.de

Summary

Pure Spectrum-Based Fault Localization (SBFL) is a well-studied statistical debugging technique that only takes a set of test cases (some failing, some passing) and their code coverage as input and produces a ranked list of suspicious program elements to help the developer identify the location of a bug that causes a failed test case. Studies show that pure SBFL techniques produce good ranked lists for small programs. However, our previous study based on the iBugs benchmark that uses the ASPECTJ repository shows that for realistic programs, the accuracy of the ranked list is not suitable for human developers. In this paper, we confirm this based on a combined empirical evaluation with the iBugs and the DEFECTS4J benchmark. Our experiments show that on average at most ∼40%, ∼80% and ∼90% of the bugs can be localized reliably within the first 10, 100 and 1000 ranked lines, respectively, in the DEFECTS4J benchmark. In order to reliably localize 90% of the bugs with the best performing SBFL metric D*, ∼450 lines have to be inspected by the developer. For human developers, this remains unsuitable, even though the results improve compared to the results for the ASPECTJ benchmark. Based on this study, we can clearly see the need to go beyond pure SBFL and take other information like information from the bug report or from version history of the code lines into consideration.

KEYWORDS:
Debugging, Fault Localization, Empirical Studies

1 | INTRODUCTION

Software is becoming the technology of choice for cost-efficiently realizing complex functionalities in a variety of systems – from critical financial infrastructures to wearable devices. As many aspects of our daily life depend on the correct operation of software, assuring its quality and prompt maintenance plays a critical role in every development process. The state of practice for checking the correctness of software and its compliance to given specifications is testing. While most test cases are still manually crafted by developers, the growing adoption by industry of automated test case generation and fuzzing techniques[1] is enabling developers to quickly discover bugs deeply buried within the complex interactions of components of a large system.

However, while tests are effective at exposing erroneous behavior, identifying the root cause behind a failure remains mostly a manual activity, requiring significant time and costs[2,3,4,5,6]. This drove a significant effort within the research community to investigate automated fault localization techniques[7]. Among them, Spectrum-Based Fault Localization (SBFL)[4] techniques exploit the evidence provided by failing and successful test cases to analyze and rank the elements of a code base according to

a degree of *suspiciousness* of their involvement with an incorrect behavior. Intuitively, the more a code element is exercised by failing test cases the more suspicious it is, i.e., the more likely it is to relate to the cause of such incorrectness. Similarly, the more a code element is exercised by successful test cases, the less suspicious it is deemed.

Beginning with Tarantula[4], a variety of SBFL approaches have been proposed, each characterized by a different suspiciousness metric. Each of these metrics lays its foundation on statistical constructions or empirical evidence of its effectiveness. If for the metric only the information from the failing and successful test executions including their coverage in the source code are used, we call these techniques pure SBFL approaches. Naish et al.[8] compared 33 SBFL approaches based on a simplified single-bug application model. Xie et al.[9,10,11] provide a theoretical evaluation of the original SBFL approaches investigated by Naish et al. and approaches that have been derived by a genetic algorithm[12]. However, most of these metrics have originally been evaluated and have subsequently been compared on relatively small-scale artifacts from the publicly available Software-artifact Infrastructure Repository (SIR)[13] (including[14,15,16,17,4,18,19,6,20,21,22,11,23]). Only recently, some SBFL techniques have been evaluated on programs[24,25] from the Defects4J benchmark[26].

In our previous work[27], we claim that an evaluation based only on small-size projects is insufficient to derive effectiveness results of SBFL techniques. Thus, we evaluated the existing pure SBFL approaches on a large scale software application namely ASPECTJ via 350 bugs from the iBugs repository[28]. ASPECTJ provides a significant challenge to pure SBFL approaches, because the system contains up to $\sim 500,000$ lines of code, depending on the development stage. Furthermore, the main success criterion used to assess such effectiveness is the localization of the fault within the top $p\%$ code elements of the produced ranking. This criterion, while easy to compute, may be practically irrelevant. In fact, if the number of code elements is large, developers would be required to inspect an impractically long list of suspects before finding the actual fault. In turn, this jeopardizes the practical effectiveness of SBFL techniques[29]. The results of our study[27] show that only 11 bugs can be detected in the best case by any of the investigated SBFL approaches after examining the top 1000 suspicious lines in the ranking. Furthermore, it would require on average 250 files that need to be inspected to discover any bug. These results demonstrate how the performance of the SBFL metrics deteriorates with the size of the code base and that the final rankings would not be acceptable for a human developer[29].

To confirm our previous results[27] and to complement the existing theoretical[9,10,11] and experimental[25,11] evaluations of SBFL metrics, this paper provides the following contributions:

- We provide an evaluation of pure SBFL approaches based on two large-scale benchmarks, namely ASPECTJ via the iBugs[28] repository and DEFECTS4J[26], where we specifically cross compare the results between the two benchmarks.
- We evaluate and compare SBFL approaches based on absolute metrics, such as the Wasted Effort (we), the number of bugs found after inspecting X ranked element ($Hit@X$), the number of files investigated (nfi), as well as area between curves (ABC) metrics which provide insights into the uncertainty of the provided ranking.
- In our evaluations, we additionally include the current state of the art SBFL metrics, namely Naish2 (Op2)[8], D*[30], the GP-evolved metric GP13[12], and Hyperbolic functions[31]. These metrics have not been covered by our previous evaluation[27].

The remainder of the paper is organized as follows. Section 2 overviews the background principles and definitions of SBFL and defines the metrics that will be evaluated and compared, as well as the criteria they have been originally evaluated on. Section 3 describes the design of our study, also summarizing our previous preliminary results[27]. Section 4 describes the ASPECTJ and DEFECTS4J case study, including the data collection process and the classification of the faults present in the code base. Section 6 concludes the work with a critical discussion of the lessons derived from the empirical evaluation in Section 5, as well as recommendations for future research directions.

2 | BACKGROUND AND DEFINITIONS

In this section, we introduce common definitions from the context of spectrum-based fault localization (SBFL) used in this paper (Section 2.1). Thereupon, we introduce the SBFL ranking metrics which are used in our experiments, starting with the traditional ranking metrics adopted from Naish et al.[8] (Section 2.2), followed by ranking metrics which have been proposed more recently (Section 2.3). Finally, we introduce common effectiveness metrics which are used to evaluate and compare different SBFL techniques with each other (Section 2.4).

2.1 | Spectrum-based Fault Localization

Using program spectra for fault localization

A *program spectrum* is a collection of information recorded during the execution of a program, characterizing its behavior w.r.t. a certain aspect[32]. While different types of program spectra have been proposed in the literature for various purposes[33], approaches to SBFL typically use *hit-spectra* which record the set of syntactical program elements that are covered during program execution. The central idea of SBFL is to use *spectra obtained from test case executions* and to additionally record for each test case whether it was successful or not. Intuitively, the higher the involvement with failing test cases and the lower the involvement with successful ones, the higher is the suspiciousness of a program element to contain a fault. The *suspiciousness score* of a program element is calculated by a *ranking metric* which yields a ranking of program elements in descending order of suspiciousness. Developers shall localize a fault by going through the list element by element, starting with the most suspicious one, based on the assumption that developers have a perfect fault understanding and can identify a fault as soon as they reach the fault location while traversing the ranking list.

In the sequel, given a program P as a set of program elements and a test suite T as a set of test cases, let $C \subseteq P$ be the set of *spectra elements*, which is defined as the union of program elements being covered when P is executed against each of the test cases in T. A ranking metric R is a function that assigns a suspiciousness score to each spectra element $c_i \in C$, denoted as $susp_R(c_i)$.

Prominent bug

A common problem of SBFL is to deal with programs containing multiple faults which may interfere with each other such that some faults cannot be localized using SBFL[34]. To overcome this limitation, DiGiuseppe and Jones[34] propose to use SBFL in an iterative process. In each step of this process, only a single fault is located and then fixed, and this step is repeated over and over again until all bugs are fixed. With this approach, the first faulty spectra element that is found, i.e., the one with the highest suspiciousness score, is the most important one, belonging to the so-called *prominent bug*. We call the respective spectra element itself the *prominent faulty spectra element*. Let $F \subseteq C$ be the set of all *faulty spectra elements*, then the following condition holds for the prominent faulty spectra element $f_{pro} \in F$:

$$\forall f_j \in F \,:\, susp_R(f_{pro}) \geq susp_R(f_j) \tag{1}$$

2.2 | Traditional SBFL Ranking Metrics

The set $C = \{c_1, \dots c_n\}$ of spectra elements obtained for executing a program P against a suite $T = \{t_1, \dots t_m\}$ of test cases may be represented by an m-by-n matrix, referred to as *coverage matrix*[30], in which each a_{ij} is a boolean value which indicates whether spectra element c_j has been executed ('true') by test case t_i or not ('false'). Consequently, a column in this matrix is a boolean vector of length m, the *coverage vector* of a spectra element c_j, indicating the test cases by which c_j is covered. In addition, the test case results yield another boolean vector of length m, called the *result vector* of executing P against T, indicating whether test case t_i causes a failure ('true') or not ('false'). The basic idea of calculating the suspiciousness score of a spectra element is that the higher the similarity of its coverage vector and the result vector, the higher is its suspiciousness. A number of specialized ranking metrics, typically adaptations of standard similarity coefficients, have been proposed in the literature. Typically, for each spectra element, a ranking metric counts the number of passed/failed involvements/non-involvements of this spectra element and combines these four numbers using a dedicated formula. An overview of the 33 traditional ranking metrics used in this paper, adopted from Naish et al.[8], is shown in Table 1. We use the notation $\langle n_{np}, n_{nf}, n_{ip}, n_{if} \rangle$ introduced by Abreu et al.[14] where the first index represents the involvement (i) or non-involvement (n) of a spectra element and the second index represents passing (p) or failing (f) test case execution. For example, n_{np} refers to the number of passing test cases in which a dedicated spectra element is not involved.

2.3 | More Recently Proposed SBFL Ranking Metrics

In the sequel, we briefly introduce more recent ranking metrics which are not included in the set of traditional ranking metrics, serving as baseline for the evaluation presented in[8] and which were not considered in our previous work[27].

Op2 or Naish2

The ranking metric Naish2 (Op2) has been developed to be optimal with respect to a single-bug application model[8]. The authors formulate two conditions which must be fulfilled by such a metric. First, when an element is involved in all failing test cases, i.e., when $n_{nf} = 0$, then the suspiciousness of this element is increasing in n_{np} or (equivalently) decreasing in n_{ip}. The intuition behind this condition is, since the buggy element of a single-bug program is executed by all failing tests, its involvement in passing test cases (n_{ip}) tends to be smaller, while its non-involvement in passing tests (n_{np}) tends to be higher. Second, when there is a failing test case in which an element is not involved ($n_{nf} > 0$), then the suspiciousness of this element is always less than any suspiciousness score of elements with $n_{nf} = 0$. Since for single-bug programs n_{nf} is always zero for the buggy element, any element with a non-zero n_{nf} value can be given a lower rank.

The simplest ranking metric which fulfills the above conditions is to assign a suspiciousness score of -1 to all elements with $n_{nf} > 0$, and n_{np} otherwise. Actually, this metric, called O in[8], can be theoretically shown to be optimal for a restricted class of single-bug programs. Another ranking metric providing the same optimality guarantees as O is Naish2 (Op2) (a.k.a. Naish2) which is defined as follows:

$$Naish2\,(Op2) := n_{if} - \frac{n_{ip}}{n_{ip}+n_{np}+1} \tag{2}$$

In our experiments, we prefer Naish2 (Op2) over O, since it performs more rationally for multiple-bug programs where n_{nf} may be non-zero for all elements. In such a case, O would assign the suspiciousness score of -1 to all elements, which is undesirable and which is avoided by Naish2 (Op2).

DStar (D*)

Wong et al.[30] derive a ranking metric from a set of requirements which are based on common intuitions. They argue that the suspiciousness assigned to a program element should be (i) proportional to the number of failed tests that cover it, (ii) inversely proportional to the number of successful tests that cover it, and (iii) inversely proportional to the number of failed tests that do not cover it. Another requirement, in the sequel referred to as (iv), is that intuition (i) is the most important one and should thus carry a higher weight.

The ranking metric derived from these requirements, named DStar (D*), is defined as:

$$D^* := \frac{n_{if}^*}{n_{ip}+n_{nf}} \tag{3}$$

where $*$ may be any positive natural number. The ranking metric is a modification of the Kulczynski coefficient[35] which is equivalent to the case $* = 1$ (D^1). The Kulczynski coefficient already embodies the above intuitions (i), (ii), (iii). Moreover, any value for $*$ greater than one increases the weight which is given to n_{if} as compared to n_{ip} and n_{nf}. This addresses intuition (iv) which states that the involvement of a program element in failing test cases should be given a higher weight than its involvement in passing test cases and its non-involvement in failing test cases, respectively.

GP-evolved ranking metrics

The traditional ranking metrics in Table 1 as well as the D* metric presented in the previous paragraph are designed by human software engineers based on general intuitions. On the contrary, Yoo[12] presents a genetic programming (GP) approach to automatically develop ranking metrics, referred to as GP-evolved ranking metrics. Starting from traditional ranking metrics, these formulae are mutated using a set of simple operators, namely addition, subtraction, multiplication, division and square root. A slightly adapted variant of the wasted effort evaluation metric (see Section 2.4) serves as fitness function for the evolutionary process.

The approach has been applied to 92 faults of four UNIX utilities taken from the SIR. The genetic programming algorithm was repeated 30 times, using a random sample of 20 faults as training data in each individual run, while reserving the 72 remaining faults for evaluation purposes. The following formula, referred to as GP13[12], represents the ranking metric obtained after 13 runs of the genetic algorithm, and it is one of the best performing metrics found by Yoo[12]:

$$GP13 := n_{if}\left(1 + \frac{1}{2n_{ip}+n_{if}}\right) \tag{4}$$

Hyperbolic metrics

Neelofar et al.[31] present a class of so-called hyperbolic metrics which are motivated by studying the contours of ranking metrics being plotted over a two-dimensional domain using n_{ip} and n_{nf} as x-axis and y-axis, respectively. The key observation is that, at

TABLE 1 Overview of all 33 examined traditional SBFL ranking metrics adopted from Naish et al.[8] and the four other metrics examined in this study (bottom right).

SBFL ranking metric	Formula
Ample	$\left\lvert \frac{n_{if}}{n_{if}+n_{nf}} - \frac{n_{ip}}{n_{ip}+n_{np}} \right\rvert$
Anderberg	$\frac{n_{if}}{n_{if}+2(n_{nf}+n_{ip})}$
Arithmetic Mean	$\frac{2n_{if}n_{np}-2n_{nf}n_{ip}}{(n_{if}+n_{ip})\cdot(n_{np}+n_{nf})+(n_{if}+n_{nf})\cdot(n_{ip}+n_{np})}$
Cohen	$\frac{2n_{if}n_{np}-2n_{nf}n_{ip}}{(n_{if}+n_{ip})\cdot(n_{np}+n_{ip})+(n_{if}+n_{nf})\cdot(n_{nf}+n_{np})}$
Dice	$\frac{2n_{if}}{n_{if}+n_{nf}+n_{ip}}$
Euclid	$\sqrt{n_{if}+n_{np}}$
Fleiss	$\frac{4n_{if}n_{np}-4n_{nf}n_{ip}-(n_{nf}-n_{ip})^2}{2n_{if}n_{nf}n_{ip}+2n_{np}n_{nf}n_{ip}}$
Geometric Mean	$\frac{n_{if}n_{np}-n_{nf}n_{ip}}{\sqrt{(n_{if}+n_{ip})\cdot(n_{np}+n_{nf})\cdot(n_{if}+n_{nf})\cdot(n_{ip}+n_{np})}}$
Goodman	$\frac{2n_{if}-n_{nf}-n_{ip}}{2n_{if}+n_{nf}+n_{ip}}$
Hamann	$\frac{n_{if}+n_{np}-n_{nf}-n_{ip}}{n_{if}+n_{nf}+n_{ip}+n_{np}}$
Hamming etc.	$n_{if}+n_{np}$
Harmonic Mean	$\frac{(n_{if}n_{np}-n_{nf}n_{ip})\big((n_{if}+n_{ip})(n_{np}+n_{nf})+(n_{if}+n_{nf})(n_{ip}+n_{np})\big)}{(n_{if}+n_{ip})\cdot(n_{np}+n_{nf})\cdot(n_{if}+n_{nf})\cdot(n_{ip}+n_{np})}$
Jaccard	$\frac{n_{if}}{n_{if}+n_{nf}+n_{ip}}$
Kulczynski1	$\frac{n_{if}}{n_{nf}+n_{ip}}$
Kulczynski2	$\frac{1}{2}\left(\frac{n_{if}}{n_{if}+n_{nf}}+\frac{n_{if}}{n_{if}+n_{ip}}\right)$
M1	$\frac{n_{if}+n_{np}}{n_{nf}+n_{ip}}$
M2	$\frac{n_{if}}{n_{if}+n_{np}+2(n_{nf}+n_{ip})}$
Ochiai	$\frac{n_{if}}{\sqrt{(n_{if}+n_{nf})\cdot(n_{if}+n_{ip})}}$
Ochiai2	$\frac{n_{if}n_{np}}{\sqrt{(n_{if}+n_{ip})\cdot(n_{np}+n_{nf})\cdot(n_{if}+n_{nf})\cdot(n_{ip}+n_{np})}}$
Overlap	$\frac{n_{if}}{\min(n_{if},n_{nf},n_{ip})}$

SBFL ranking metric	Formula
Rogers & Tanimoto	$\frac{n_{if}+n_{np}}{n_{if}+n_{np}+2(n_{nf}+n_{ip})}$
Rogot1	$\frac{1}{2}\left(\frac{n_{if}}{2n_{if}+n_{nf}+n_{ip}}+\frac{n_{np}}{2n_{np}+n_{nf}+n_{ip}}\right)$
Rogot2	$\frac{1}{4}\left(\frac{n_{if}}{n_{if}+n_{ip}}+\frac{n_{if}}{n_{if}+n_{nf}}+\frac{n_{np}}{n_{np}+n_{ip}}+\frac{n_{np}}{n_{np}+n_{nf}}\right)$
Russell & Rao	$\frac{n_{if}}{n_{if}+n_{nf}+n_{ip}+n_{np}}$
Scott	$\frac{4n_{if}n_{np}-4n_{nf}n_{ip}-(n_{nf}-n_{ip})^2}{(2n_{if}+n_{nf}+n_{ip})\cdot(2n_{np}+n_{nf}+n_{ip})}$
Simple Matching	$\frac{n_{if}+n_{np}}{n_{if}+n_{nf}+n_{ip}+n_{np}}$
Sokal	$\frac{2(n_{if}+n_{np})}{2(n_{if}+n_{np})+n_{nf}+n_{ip}}$
Sørensen-Dice	$\frac{2n_{if}}{2n_{if}+n_{nf}+n_{ip}}$
Tarantula	$\frac{\frac{n_{if}}{n_{if}+n_{nf}}}{\frac{n_{if}}{n_{if}+n_{nf}}+\frac{n_{ip}}{n_{ip}+n_{np}}}$
Wong1	n_{if}
Wong2	$n_{if}-n_{ip}$
Wong3	$\begin{cases} n_{if}-n_{ip} & \text{if } n_{ip}\le 2 \\ n_{if}-\left(2+\frac{1}{10}(n_{ip}-2)\right) & \text{if } 2<n_{ip}\le 10 \\ n_{if}-\left(2.8+\frac{1}{1000}(n_{ip}-10)\right) & \text{otherwise} \end{cases}$
Zoltar	$\frac{n_{if}}{n_{if}+n_{nf}+n_{ip}+\frac{10000n_{nf}n_{ip}}{n_{if}}}$

Naish2 (Op2)	$n_{if}-\frac{n_{ip}}{n_{ip}+n_{np}+1}$
DStar (D*)	$\frac{n_{if}^{*}}{n_{ip}+n_{nf}}$
GP13	$n_{if}\left(1+\frac{1}{2n_{ip}+n_{if}}\right)$
Hyperbolic	$\frac{1}{K_1+\frac{n_{nf}}{n_{if}+n_{nf}}}+\frac{K_3}{K_2+\frac{n_{ip}}{n_{if}+n_{ip}}}$

the bottom right and the top left, the contours of hyperbolas $\frac{1}{n_{nf}}+\frac{1}{n_{ip}}$ are like the contours of ranking metrics which are known to be optimal for single-bug programs (Naish2 (Op2), see Naish et al.[8]) and for programs with only deterministic bugs causing a failure whenever they are executed (O^d, see Naish et al.[36]), respectively. Thus, Neelofar et al. argue that hyperbolas can be adapted to either of the two extremes and, more importantly, to any setting between the two extreme cases, i.e., the usual case in practice that a program has multiple bugs of which at least some are non-deterministic. To that end, the simple formula $\frac{1}{n_{nf}}+\frac{1}{n_{ip}}$ is adjusted and parameterized as follows:

$$Hyperbolic := \frac{1}{K_1+\frac{n_{nf}}{n_{if}+n_{nf}}}+\frac{K_3}{K_2+\frac{n_{ip}}{n_{if}+n_{ip}}} \tag{5}$$

The adjustment compared to the formula $\frac{1}{n_{nf}}+\frac{1}{n_{ip}}$ is that both n_{nf} and n_{ip} values are scaled to the range $[0,1]$. Furthermore, the parameters K_1, K_2 and K_3 can be used to make contours flatter or steeper. Neelofar et al.[31] propose a machine learning approach based on genetic programming and simulated annealing for optimizing the parameter values on a given training data set. In contrast to the GP-evolved ranking metrics introduced earlier in this section, only three numeric parameter values need to be evolved instead of evolving entire formulae as proposed by Yoo[12].

2.4 | Common Effectiveness Metrics for Evaluating 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.

Preliminaries

Common effectiveness metrics evaluate SBFL techniques by examining the position of the faulty element in the suspiciousness ranking. However, the ranking position may be non-deterministic if multiple ranked elements have the same suspiciousness score. Since SBFL techniques do not provide additional information on how to rank such elements, all elements with the same score are randomly ordered. If there are multiple elements which are equally suspicious 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:

$$best\_rank_R(c_j) = \left|\{c_i \in C \mid susp_R(c_i) > susp_R(c_j)\}\right| + 1 \tag{6}$$

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

$$worst\_rank_R(c_j) = \left|\{c_i \in C \mid susp_R(c_i) \geq susp_R(c_j)\}\right| \tag{7}$$

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

$$avg\_rank_R(c_j) = \frac{1}{2}\left(best\_rank_R(c_j) + worst\_rank_R(c_j)\right) \tag{8}$$

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

Wasted Effort (WE)

As already mentioned, one of the basic assumptions of SBFL is that, in order to locate a fault, a developer has to inspect all the program elements that are ranked higher than the faulty element. The number of inspected non-faulty elements may be thus defined as the wasted effort for the developer. To date, the wasted effort is usually put in relation to the total number of ranked elements, turning it into a relative metric which is defined as follows:

$$min\_we_R(c_j) = \frac{best\_rank_R(c_j)-1}{|C|} \tag{9}$$

$$max\_we_R(c_j) = \frac{worst\_rank_R(c_j)-1}{|C|} \tag{10}$$

Please note that min_we_R and max_we_R are defined for faulty spectra elements, and not for the ranking metric R itself. Thus, the results obtained from the wasted effort metrics above need to be aggregated in order to calculate an effectiveness score for a specific ranking metric. This can be achieved by an aggregation function a, for example, by taking the average values of the wasted effort obtained for all faulty spectra elements:

$$min\_we_a(R) = a(\{min\_we_R(f_j) \mid f_j \in F\}) \tag{11}$$

$$max\_we_a(R) = a(\{max\_we_R(f_j) \mid f_j \in F\}) \tag{12}$$

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

Relative Proportion of Bugs Localized (rPBL)

Another commonly used effectiveness metric is the proportion of bugs localized when examining a certain *percentage of spectra elements*. This effectiveness metric is defined for a ranking metric R and directly produces a score which can be compared to the score of other ranking metrics. Let $p \in [0, 1]$ be the percentage of inspected program elements, then two variants of the rPBL effectiveness metric are defined as follows:

$$min\_pbl_p(R) = \frac{\left|\{f \in F \mid max\_we_R(f) < p\}\right|}{|F|} \tag{13}$$

$$max\_pbl_p(R) = \frac{\left|\{f \in F \mid min\_we_R(f) < p\}\right|}{|F|} \tag{14}$$

Absolute Proportion of Bugs Localized (aPBL)

We also use the proportion of bugs localized when examining a certain *number of spectra elements*. Like the relative version, this effectiveness metric is defined for a ranking metric R and directly produces a score which can be compared to the score of other ranking metrics. Let $n \in \mathbb{N}$ be the number of inspected program elements, then the two variants of the aPBL effectiveness metric are defined as follows:

$$min\_pbl_n(R) = \frac{\left|\{f \in F \mid worst\_rank_R(f) \leq n\}\right|}{|F|} \tag{15}$$

$$max\_pbl_n(R) = \frac{\left|\{f \in F \mid best\_rank_R(f) \leq n\}\right|}{|F|} \tag{16}$$

Note that it will be clear from the context whether the relative or absolute version of the metric is being used.

Bugs Localized in the first X Elements (Hit@X)

A crucial issue w.r.t. traditional SBFL effectiveness metrics which has been raised by Parnin and Orso[29] is that developers only investigate a certain absolute number of ranked elements before giving up and using alternative debugging methods. Lucia et al.[20] address this issue by introducing a new 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 authors have chosen 10 as threshold for the metric. For our study, we generalize the metric as follows to have a varying threshold of X ranked elements:

$$min\_Hit@X(R) = \left|\{f \in F \mid worst\_rank_R(f) \leq X\}\right| \tag{17}$$

$$max\_Hit@X(R) = \left|\{f \in F \mid best\_rank_R(f) \leq X\}\right| \tag{18}$$

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

2.5 | Granularity Level of Program Elements

Note that we did not specify the exact nature of program elements up to this point. While the most popular fine-grained granularity level is statements, there are studies examining the capabilities of SBFL for coarser levels of granularity such as blocks or methods[37,38,39]. A recent study by Kochhar et al.[40], surveying 386 software engineering practitioners, found that statement, block and method level granularity are preferred by practitioners, while coarser granularity levels like class or even component level are mostly deemed not sensible in practice. While the surveyed practitioners slightly preferred method level granularity (51.81%) over statement and block level granularity (50.00% and 44.30%, respectively), the study states that "[t]here is no clear winner among these three granularity levels [...]".

In our view, statement level granularity has the benefit over coarser granularity levels of allowing for a more reasonably founded estimate of the *actual* effort a developer has to put into localizing a fault with the aid of SBFL techniques. Coarser program elements like blocks or methods may consist of heavily varying numbers of statements that, in turn, make up constructs of greatly varying complexity. Thus, the actual effort for a developer to process a coarser program element will differ greatly between different elements.

As an example, imagine two rankings consisting of, e.g., methods. Assume that the first ranking contains 10 very small methods, while the second one contains 10 methods of very large size. If a faulty method is ranked in 10th place in both rankings, they are identical to popular evaluation metrics like, e.g., wasted effort or $Hit@X$ as defined above, but the first list of methods will be much more acceptable to a developer than the second.

Due to the above reason, as well as to keep this study's experiments and results consistent with our previous study[27], we decided to adhere to statement level granularity in our experiments.

3 | STUDY DESIGN

3.1 | Previous Findings and Research Questions

Previous experiments[15,17,20] with small-sized programs as well as an empirical study of ranking-based automatic debugging techniques with 68 developers[29] indicate several problems w.r.t. the practicability of SBFL techniques for large-scale software systems. These problems may be summarized as follows:

- SBFL techniques reveal faulty statements only after inspecting a large number of lines or code elements[29].

- SBFL techniques often assign the same suspiciousness scores for multiple lines or code elements[20].
- Users of SBFL techniques do not inspect in the order provided by the ranked list[2,29].
- SBFL techniques require a large number of failing test cases to accurately reveal a fault[15,17].

The goal of this paper is to investigate if these problems really exist for a complex real-world software system. In the following, we will have a closer look at each of the aforementioned problems and derive the research questions of our study.

SBFL techniques reveal faulty program statements only after inspecting a large number of lines or code elements

Strictly interpreting the traditional evaluation strategies for measuring the effectiveness of SBFL techniques (see Section 2.4) – particularly the relative metrics such as the Wasted Effort (WE) metric (see Eq. 9 and 10) or the Proportion of Bugs Localized (PBL) metric (see Eq. 13 and 14) – developers need to inspect several thousand lines of code 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[29].

RQ$_1$ *What is the absolute SBFL effectiveness?* (Section 5.1)

SBFL techniques often assign the same suspiciousness scores for multiple lines or code elements

SBFL ranking metrics use only four numbers $\langle n_{np}, n_{nf}, n_{ip}, n_{if} \rangle$ to calculate the suspiciousness score of a program element (cf. Section 2). The sum of them corresponds to the number of available execution traces, i.e., the number of test cases available in the given test suite. If only a small number of test cases is available, the number of possible different inputs to SBFL algorithms decreases drastically and the amount of different suspiciousness scores is limited. As a consequence, multiple program elements will share the same suspiciousness score.

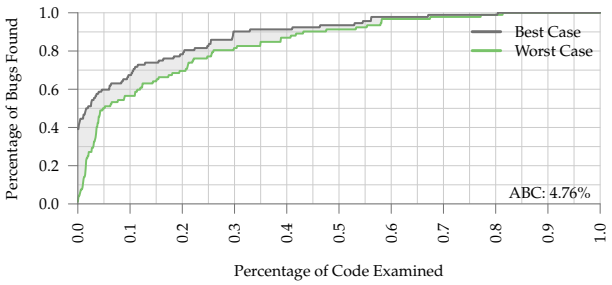


FIGURE 1 Results for the rPBL metric for Tarantula for the SIR programs `flex`, `gzip`, `grep`, and `sed`.

The real drawback of multiple program elements having the same suspiciousness scores arises when producing an ordered list of program elements which is to be inspected by developers. If multiple program elements share the same suspiciousness, those elements cannot be distinguished, and the actual physical order can only be chosen randomly. This randomness does not have a big influence on the developer's investigation if only a few program elements share the same suspiciousness, but the approach becomes increasingly inaccurate if this is the case for more than hundreds to thousands of program elements. As an example, Fig. 1 shows that even for relatively small programs like `flex`, `gzip`, `grep` and `sed`, there is a remarkable difference between best and worst case rankings, and this uncertainty may accentuate for larger programs.

RQ$_2$ *What is the uncertainty in the assigned suspiciousness scores?* (Section 5.2)

Users of SBFL techniques do not inspect in the order provided by the ranked list

Parnin and Orso[29] have shown in their study that developers do not linearly follow the ranking produced by SBFL. Instead, they use the most suspicious statements as starting points for their investigation and then *search* for the actual fault location in the surrounding method, class, or file. 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 5.3)

SBFL techniques require a large number of failing test cases to accurately reveal a fault

The results in Abreu et al.[15] show that on the SIR benchmark[13] even a small number of failing test cases ($n_{nf} + n_{if}$) provides a reasonable fault localization performance. However, the study also recommends that more related failing test cases (specifically the ones that trigger the faulty lines n_{if}) are always better.

RQ$_4$ *What is the relation between the number of related and unrelated failing test cases and SBFL's accuracy?* (Section 5.4)

3.2 | Study Subject

As in our previous study[27], we use ASPECTJ[41] which is a compiler system that provides developers with an aspect-oriented extension to the Java programming language. Additionally, we examine the DEFECTS4J benchmark[26], a collection of well-prepared, real bugs taken from 6 different open-source Java projects: *JFreechart*, *Closure*, *Apache commons-lang*, *Apache commons-math*, *Joda-Time* and *Mockito* (cf. Section 4). From these six programs, Closure is also a compiler similar to ASPECTJ. However, based on the other 5 programs, we improve diversity among the study subjects.

3.3 | Study Protocol

For our study, we use the available data for ASPECTJ that was gathered[27], and for DEFECTS4J, we automatically mined the lines related to the bugs. We do a comparison of all 33 used SBFL techniques summarized by Naish et al.[8] and four additional pure SBFL metrics (cf. Section 2.3). All examined SBFL metrics are given in Table 1. The goal of the descriptive evaluation is to answer our research questions ($\mathbf{RQ}_1$-$\mathbf{RQ}_4$). Specifically, we will use the absolute wasted effort metric and the *Hit@X* metric (Eq. 17 and 18) to answer $\mathbf{RQ}_1$. To answer $\mathbf{RQ}_2$ and $\mathbf{RQ}_3$, we use and extend our metrics developed in our previous studies[27] which will be detailed in the following sections. To answer $\mathbf{RQ}_4$, we use the relative wasted effort metric (Eq. 9 and 10).

3.3.1 | (absolute) Area Between Curves (ABC)

Traditional effectiveness evaluations always define a metric for the best and the worst case (and implicitly, due to the uniform distribution, also an average case). If the best and worst case diverge, this means that there are ranked elements sharing the same suspiciousness score, which in turn implies that parts of the ranking are randomly ordered. To measure the divergence of a given best/worst case metric, the area between the best- and worst-case curves can be leveraged. As an extension of the ABC metric[27], we propose a variation of this metric which can be restricted to a specified number of ranked elements that are to be considered by the metric instead of being based on the entire rankings.

Since $M_{worst} \leq M_{best}$ holds in all points for a metric M, the *area between the curves up to rank r* is computed by:

$$ABC_M(r) = AUC_{M_{best}}(r) - AUC_{M_{worst}}(r), \tag{19}$$

where $AUC_{M'}(r)$ is the *area under the curve up to rank r* for a metric M':

$$AUC_{M'}(r) = \frac{1}{n} \cdot \left(\frac{y_0}{2} + \sum_{i=1}^{n-1} y_i + \frac{y_n}{2} \right), \tag{20}$$

for a set of equidistant points (x_i, y_i), $i \in [0, n]$, $n \in \mathbb{N}^+$ and $x_0 = 0$ and $x_n = r$. Since we will use the PBL metric in our experiments, we define the set of support points as

$$\begin{aligned} x_i &= \tfrac{i \cdot r}{n},\ y_i = max\_pbl_{x_i}(R) \quad \text{for } M' = M_{best}, \text{ and} \\ x_i &= \tfrac{i \cdot r}{n},\ y_i = min\_pbl_{x_i}(R) \quad \text{for } M' = M_{worst}. \end{aligned}$$

As in[27], the ABC value is presented as percentage of the maximum area that can be achieved by a completely diverging best and worst case. This area is restricted by the maximum amount of bugs that is found by an SBFL metric R within the first r ranking elements. We will show the upper bound of this area using a horizontal dashed line in the plots which marks the value $y_n = max\_pbl_{x_n=r}(R)$.

3.3.2 | Number of Files Investigated (NFI)

To assess the effectiveness of pointing a developer to the right places when following a 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. The metric is 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:

$$min\_nfi_R(c_j) = \left| \{ file(c_k) \mid worst\_rank_R(c_k) < best\_rank_R(c_j) \} \cup \{ file(c_j) \} \right| \tag{21}$$

$$max\_nfi_R(c_j) = \left| \{ file(c_k) \mid worst\_rank_R(c_k) \leq worst\_rank_R(c_j) \} \right| \tag{22}$$

4 | DATA COLLECTION & PREPARATION: ASPECTJ AND DEFECTS4J

4.1 | ASPECTJ

In our previous study[27], we mined all 350 bugs from the iBugs repository[28] 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 $\sim 200,000$ to $\sim 500,000$ lines of code (data taken from Openhub[42]). To evaluate the quality of the different SBFL rankings, the source code changes were inspected and each line was manually classified as either *buggy* or *healthy*[27].

TABLE 2 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

One major result of our previous study[27] was that from the 350 buggy versions, only 88 were applicable for SBFL and 262 buggy versions needed to be removed from the dataset because:

- 86 buggy versions were classified as *enhancement* and not as a bug.
- 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).
- 65 bugs either had only faulty lines classified with low confidence or did not produce any faulty execution traces due to compile time or runtime errors. Examples of runtime errors are environment issues based on hardware constraints that lead to `OutOfMemoryErrors` that were resolved by adding appropriate try-catch clauses. Another example are concurrency issues that are fixed by adding a `synchronized` modifier to a method.

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 2 shows the number of bugs and how the set of bugs was reduced through the various stages of data preparation. From all 350 bugs in the benchmark, $\sim$25% were applicable for SBFL and $\sim$15% were actually involved in at least one test case.

4.2 | DEFECTS4J

As a second benchmark for our experiments, we used the 6 buggy open-source projects in the DEFECTS4J benchmark[1] which currently contains a total amount of 395 bugs. The DEFECTS4J benchmark includes a fixed change set for each bug that *only includes changes related to the error*, i.e., no refactorings or other changes to the source code that do not serve to fix the respective bug are performed. Thus, the benchmark is more suitable for fault localization experiments[43] and provides a cleaner environment than the ASPECTJ benchmark, which is on the one hand messy, but on the other hand it may provide a more realistic picture of real bugs. From the 395 Bugs present in the DEFECTS4J benchmark, only 370 were found to be applicable for SBFL and are included in our experiments. The removed bugs only contain bug-related lines that are not executable, i.e., lines that are not covered by the SBFL rankings. Please note that the study by Pearson et al.[25] just uses 310 bugs for their experiments. The number of total bugs for each project is shown in Table 3.

TABLE 3 Bugs in the DEFECTS4J benchmark.

project	size in kloc	number of bugs	applicable bugs
JFreechart (Chart)	96	26	25
Closure compiler (Closure)	90	133	126
Apache commons-lang (Lang)	22	65	54
Apache commons-math (Math)	85	106	104
Joda-Time (Time)	28	27	24
Mockito (Mockito)	68	38	37
total		395	370

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

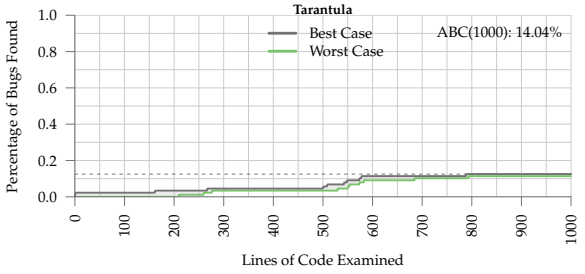
5 | EXPERIMENTAL RESULTS

5.1 | RQ$_1$: What is the absolute SBFL performance?

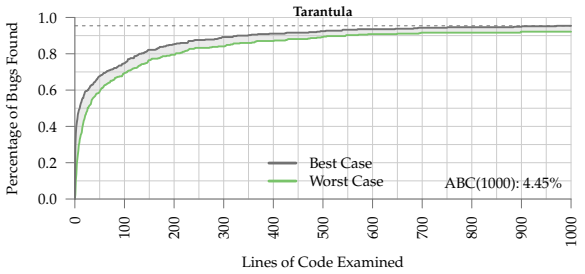
This research question examines the absolute bug localization performance of pure SBFL techniques, i.e., their usefulness for human developers, in a sense. Section 5.1.1 and Section 5.1.2 examine the general performance of many existing SBFL metrics on the ASPECTJ and DEFECTS4J benchmarks with the *absolute wasted effort* and the *Hit@X* metric, respectively, while Section 5.1.3 examines the performance of selected state of the art SBFL metrics in more detail.

5.1.1 | Absolute Wasted Effort

Fig. 2 shows the percentage of bugs found in the ASPECTJ and DEFECTS4J benchmarks using the Tarantula ranking metric after examining 1000 lines of code. We consider a bug to be found by a developer as soon as the first line related to the bug is encountered in the line ranking. Arguably, this is a rather unreasonable assumption to make, as, e.g., found by Parnin and Orso[29]. It attributes a perfect understanding of the code to the developer, such that (s)he is able to recognize the bug after seeing the first bug related line. Note that this is the best case scenario which usually does not reflect reality.



(a) Localization performance of all 88 examined bugs in ASPECTJ.



(b) Localization performance of all 370 examined bugs in DEFECTS4J.

FIGURE 2 Absolute bug localization performance within the first 1000 ranked elements for both benchmarks.

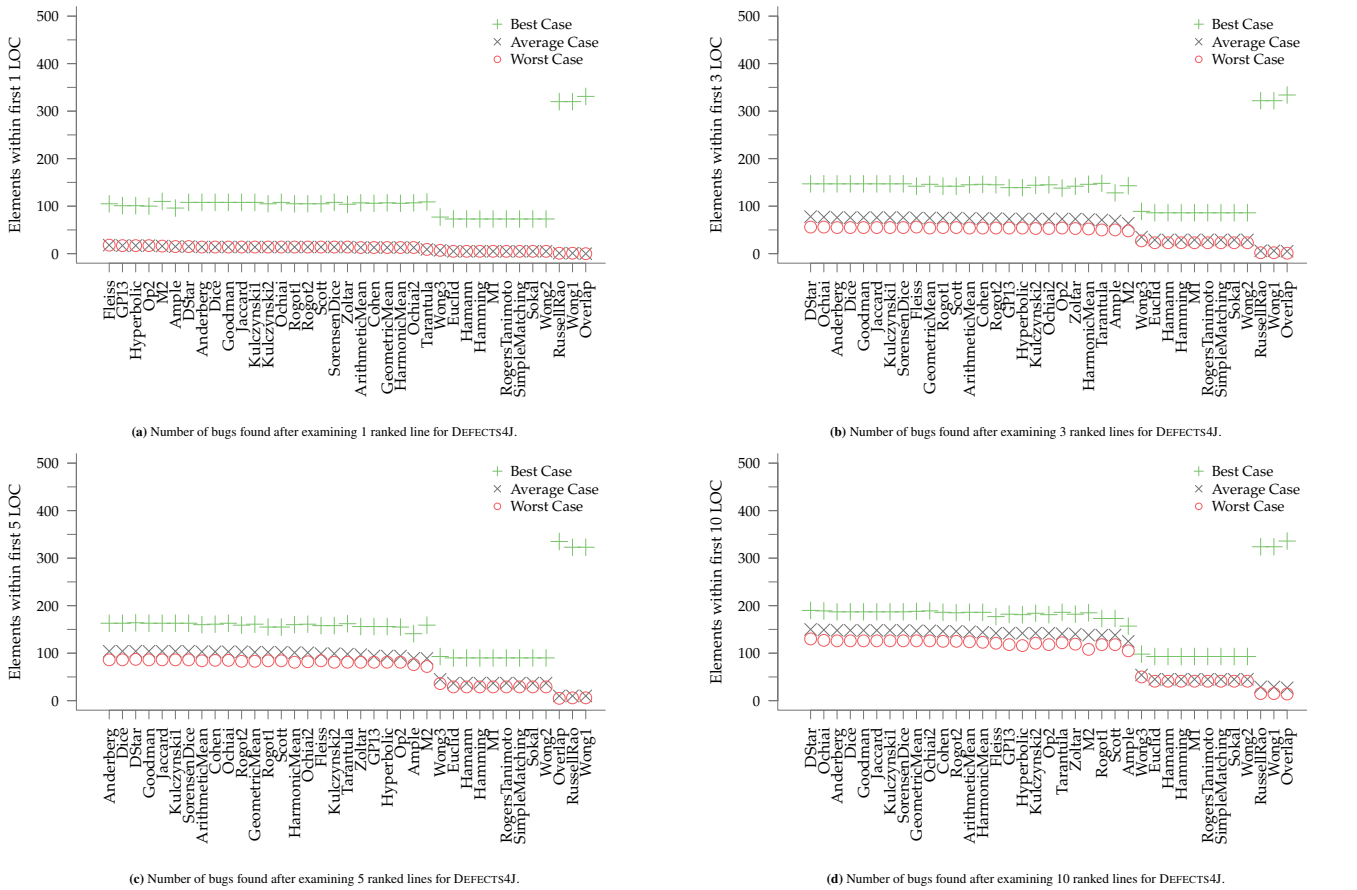
We can see that the respective plot for ASPECTJ in Fig. 2a shows a significantly lower localization performance than the plot for DEFECTS4J in Fig. 2b. For ASPECTJ, we only encounter slightly more than 10% bugs within the first 1000 ranked elements, whereas we are able to find approximately 70% of bugs already in the first 100 ranked elements and around 90% of bugs in the first 1000 ranked elements for the DEFECTS4J benchmark.

This shows that SBFL's (here, in particular: Tarantula's) fault localization capability is strongly tied to the buggy programs under consideration. For ASPECTJ, we usually have multiple tests that fail, but some are not actually related to the bug under consideration, i.e., after fixing the bug, some (and sometimes even all) of the failing tests continue to fail. In the DEFECTS4J benchmark, only failing test cases related to the respective bug are included, i.e., after fixing the bug, all tests succeed. This is the biggest cause of the existing differences in bug localization performance that we observe. For the ASPECTJ benchmark, failing test cases that are unrelated to the respective bug generally mislead the SBFL techniques and negatively distort the results.

We also strengthen our results from our previous studies[27] that the performance of SBFL metrics is not tied to the amount of test cases, since the bugs in the ASPECTJ benchmark have an average number of $\sim 1,500$ test cases each, with ~ 134 failing test cases, while the bugs in the DEFECTS4J benchmark only contain an average number of ~ 735 test cases and ~ 3.4 failing test cases. A big test suite is not enough to ascertain good SBFL performance. Rather, it is test case quality that matters.

A second but less important difference is the different ranking sizes. Meaning that, on average, there are significantly more elements in the SBFL rankings for the entities in ASPECTJ than there are for DEFECTS4J. For the ASPECTJ benchmark, the ranking sizes range from around 15,000 elements to around 210,000 elements. The mean ranking size is $\sim 126,000$ elements and the median ranking size is $\sim 187,000$ elements. For the DEFECTS4J benchmark, the size of the examined rankings differs from around 2,000 elements to around 55,000 elements based on the projects and the specific bugs themselves. The mean ranking size is $\sim 28,000$ elements and the median ranking size is $\sim 31,500$ elements.

The ASPECTJ benchmark is not very suited to successfully perform SBFL techniques on. The rankings are very large (up to 210,000 elements), and noise is introduced by failing test cases that are not related to the bugs that are examined. Only $\sim 10\%$ of the bugs are localized within the first 1,000 ranked elements. For the DEFECTS4J benchmark, on the other hand, we are able to localize $\sim 90\%$ of the bugs within the first 1,000 ranked elements.



(a) Number of bugs found after examining 1 ranked line for DEFECTS4J.

(b) Number of bugs found after examining 3 ranked lines for DEFECTS4J.

(c) Number of bugs found after examining 5 ranked lines for DEFECTS4J.

(d) Number of bugs found after examining 10 ranked lines for DEFECTS4J.

FIGURE 3 *Hit*@1/3/5/10 metric for all SBFL techniques for DEFECTS4J.

5.1.2 | *Hit@X*

Fig. 3 and 4 show the *Hit*@1/3/5/10 metric (Eq. 17 and 18) for all examined SBFL ranking metrics.

In each plot, the vertical axis represents the number of bugs found after examining the first 1/3/5/10 ranked elements, respectively, produced by each SBFL ranking metric, and each plot marks the best-, average-, and worst-case for each ranking metric. Fig. 4 shows the plot for the ASPECTJ benchmark, and Fig. 3 shows the respective plots for DEFECTS4J. For ASPECTJ, the plots for the four *Hit@X* metrics were identical, so that it made sense to combine them into a single plot. Note that the plots for ASPECTJ and DEFECTS4J are not directly comparable, since the number of examined bugs in the benchmarks is very different from each other (88 bugs for ASPECTJ, 370 bugs for DEFECTS4J).

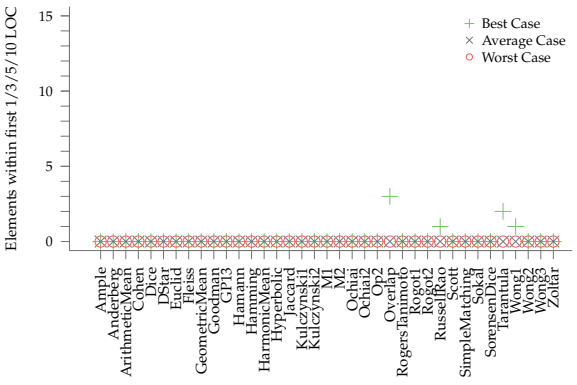
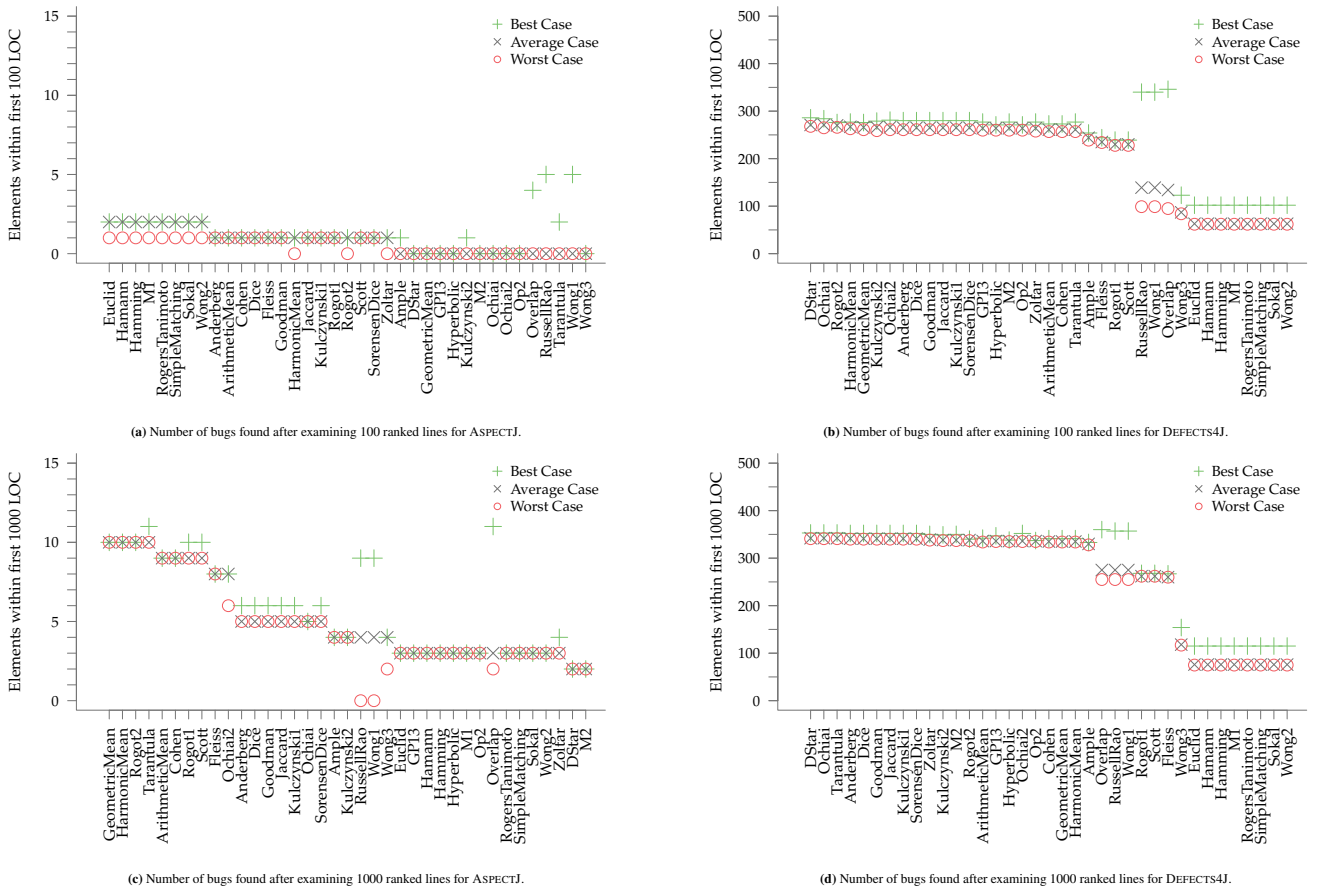


FIGURE 4 *Hit*@1/3/5/10 metric for all SBFL techniques for ASPECTJ.

The plots show that all examined SBFL techniques deliver better results on the DEFECTS4J benchmark with most of them being able to find ~150 bugs (~40%) on average within the first 10 ranked elements (Fig. 3d). For ASPECTJ, finding a bug in the first 10 lines is impossible with most of the SBFL techniques and depends on sheer luck for the remaining ones. At most 3 out of 88 bugs (~3.4%) may be found in the *best* case using the Overlap metric. On the other hand, for DEFECTS4J, even when only examining the *first* ranked element for each bug in the benchmark (Fig. 3a), the decently performing SBFL metric Ochiai2 finds 13 out of 370 bugs (~3.5%) in the *worst* case, i.e., independent of the ordering of elements with identical suspiciousness scores. The Fleiss metric even ranks 18 bugs (~ 4.9%) within the first ranked line in the worst case. In comparison to the ASPECTJ benchmark, the Overlap metric locates 321 out of 370 DEFECTS4J bugs (~ 86.8%) to the first line of the ranking in the best



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

(b) Number of bugs found after examining 100 ranked lines for DEFECTS4J.

(c) Number of bugs found after examining 1000 ranked lines for ASPECTJ.

(d) Number of bugs found after examining 1000 ranked lines for DEFECTS4J.

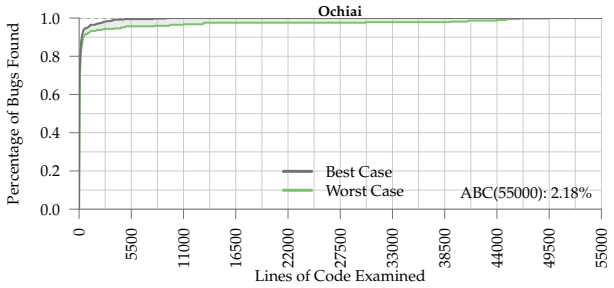
FIGURE 5 *Hit*@100/1000 metric for all SBFL techniques for ASPECTJ and DEFECTS4J.

case. This, however, depends on sheer luck, once again. In practice, the Overlap metric is simply too unreliable to be used by developers (cf. Section 5.2).

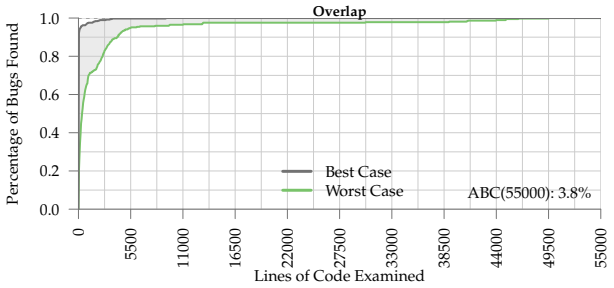
Fig. 5 shows the *Hit*@100 and *Hit*@1000 metrics for all examined SBFL ranking metrics for both ASPECTJ and DEFECTS4J. For the DEFECTS4J benchmark, the group of best performing metrics are able to reliably find close to 300 out of 370 bugs, i.e., ∼80% of the bugs within the first 100 lines and close to 350 bugs (∼95%) within the first 1000 lines. While for the ASPECTJ benchmark, even finding more than 10 out of 88 bugs within the first 1000 lines is only achieved by the Tarantula metric in the best case. Note that looking at 1000 or even just 100 ranked elements will usually not even be feasible for a human developer[29].

For the DEFECTS4J benchmark and based on the plots (specifically Fig. 5b, Fig. 5d and additionally Fig. 3), the examined SBFL techniques can be roughly grouped into 4 similarly performing groups in terms of bug localization performance. The best performing group includes well-known and established ranking metrics like Tarantula, Jaccard, Ochiai and Op2. All of these metrics perform approximately equally well and it cannot be reasonably decided whether one is significantly better than the other based on their usefulness for the developer. Out of all the examined metrics, DStar (cf. Section 5.1.3) performs best overall, but the difference to the other metrics in the group is hardly noticeable. The second group of metrics contains the three ranking metrics Wong1, RussellRao, and Overlap. For group three, which includes the ranking metrics Rogot1, Fleiss, and Scott we actuallally see a shift in performance between the *Hit*@100 and *Hit*@1000 results. For *Hit*@100 they outperform group 2 and for *Hit*@1000 they are worse. All remaining ranking metrics, starting with the ranking metric Wong3, belong to group four. Fig. 6 shows plots of four metrics that can be considered representative for each group, based on the rough shape of their plots. As the plots look very similar for metrics in the same group, we decided to only show the ones for Ochiai, Overlap, Scott and RogersTanimoto, as an example.

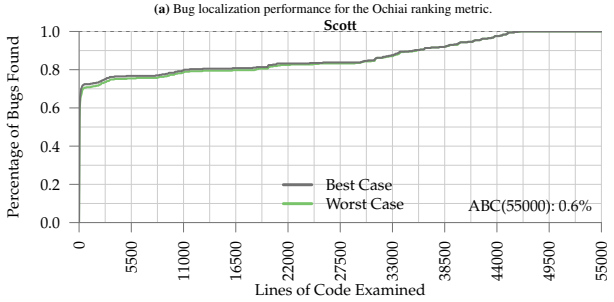
Even with the addition of new, supposedly superior SBFL metrics, the results for the ASPECTJ benchmark do not improve over the results found in our previous studies[27]. The results for the DEFECTS4J benchmark are more promising with most of the examined SBFL metrics being able to find ∼40% of the bugs in the first 10 ranked elements and ∼80% of the bugs within the first 100 ranked elements. Within the first 1000 ranked elements, even ∼90% of the bugs can be localized.



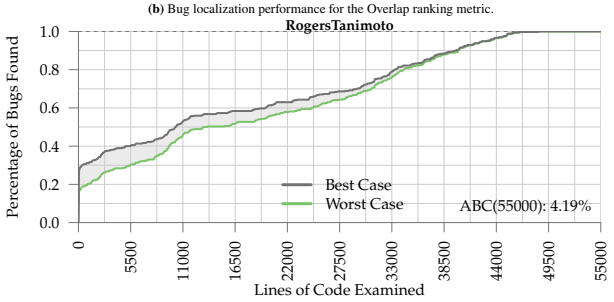
(a) Bug localization performance for the Ochiai ranking metric.



(b) Bug localization performance for the Overlap ranking metric.

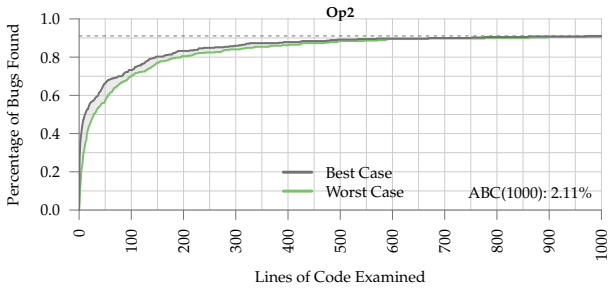


(c) Bug localization performance for the Scott ranking metric.

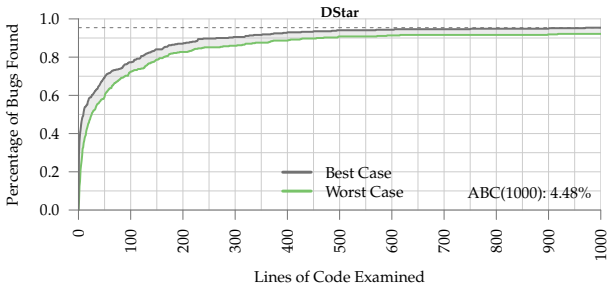


(d) Bug localization performance for the RogersTanimoto ranking metric.

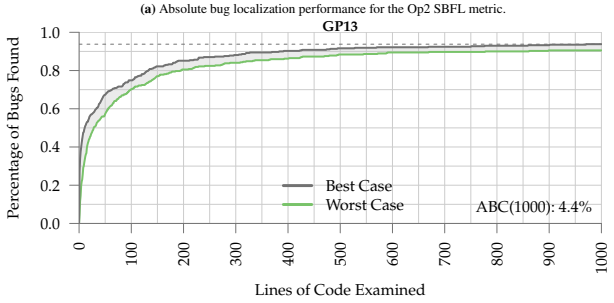
FIGURE 6 Absolute bug localization performance in the DEFECTS4J benchmark for different SBFL techniques.



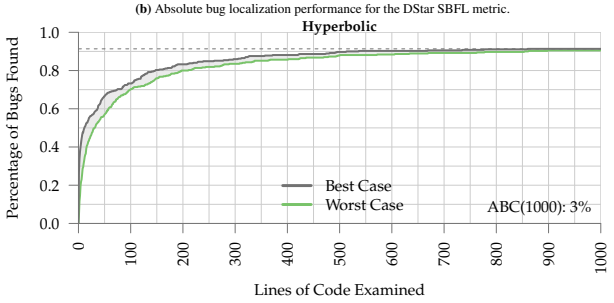
(a) Absolute bug localization performance for the Op2 SBFL metric.



(b) Absolute bug localization performance for the DStar SBFL metric.



(c) Absolute bug localization performance for the GP13 SBFL metric.

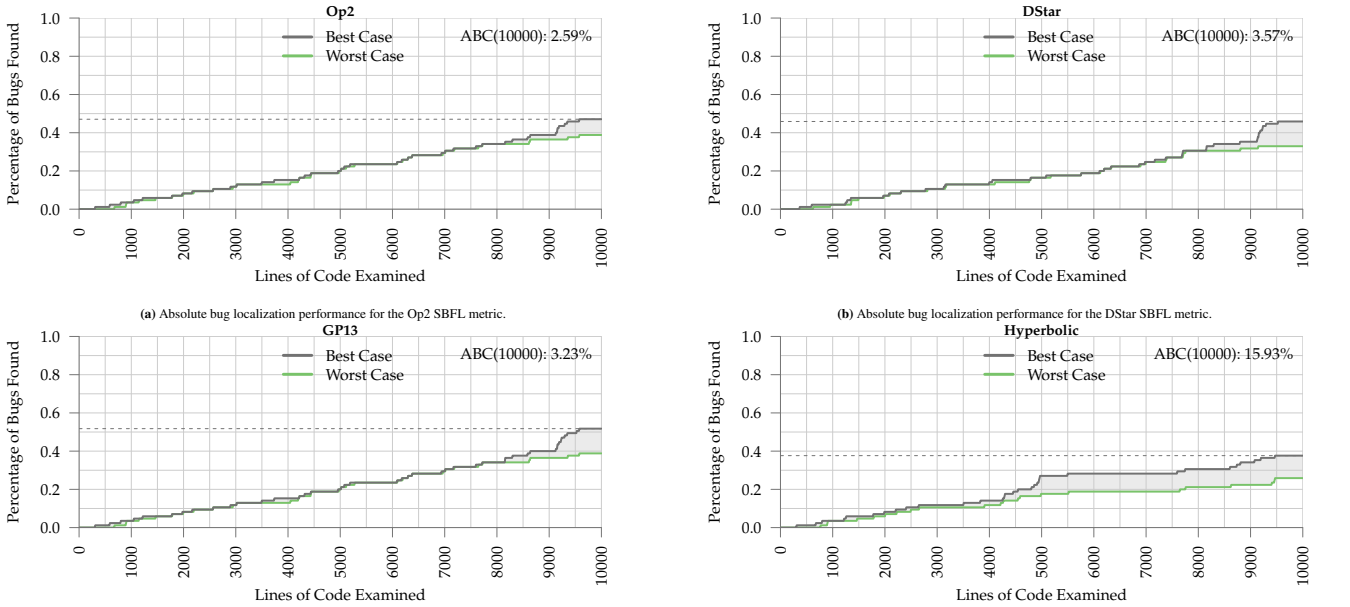


(d) Absolute bug localization performance for the Hyperbolic SBFL metric.

FIGURE 7 Absolute bug localization performance for chosen metrics on 370 applicable bugs in the DEFECTS4J benchmark within the first 1000 lines.

5.1.3 | Performance of specific state of the art SBFL metrics

In this section, we examine the absolute bug localization performance of 4 selected, pure SBFL metrics (cf. Section 2.3) that were not included in our previous work[27]. These metrics were designed in an effort to find SBFL metrics that outperform other, already existing pure SBFL metrics. This section aims to examine the practical value of these metrics for human developers and to measure the gained improvements, if any.



(a) Absolute bug localization performance for the Op2 SBFL metric.

(b) Absolute bug localization performance for the DStar SBFL metric.

(c) Absolute bug localization performance for the GP13 SBFL metric.

(d) Absolute bug localization performance for the Hyperbolic SBFL metric.

FIGURE 8 Absolute bug localization performance for chosen metrics on 88 applicable bugs in the ASPECTJ benchmark within the first 10,000 lines.

Naish2 (Op2)

The Op2 metric was designed to be single-bug optimal. The plots in Fig. 3 and Fig. 5 show that the Op2 metric is in the best performing group for the DEFECTS4J benchmark. For the ASPECTJ benchmark, however, it does not perform that well, finding no bugs at all in the first 100 ranked elements and finding only 3 bugs in the first 1000 ranked elements, while, e.g., Tarantula is able to locate at least 10 bugs within the first 1000 ranked elements. Fig. 7a shows the absolute bug localization performance in the first 1000 ranking elements for the DEFECTS4J benchmark using the Op2 metric. Fig. 8a shows the bug localization performance of the Op2 metric for the ASPECTJ benchmark within the first 10,000 lines. Note that we increase the range to 10,000 ranked elements for the plots for the ASPECTJ benchmark, since the percentage of localized bugs in the first 1,000 ranked elements is very low and, thus, plots within this range are very hard to read and to compare.

DStar (D*)

The DStar (D*) metric is a modification of the Kulczynski coefficient and was shown to be a very effective fault localization metric. For the DEFECTS4J benchmark, the plots in Fig. 3, Fig. 5b and Fig. 5d confirm this, as the DStar metric locates the most bugs in the first 10 lines and the first 100 lines, and it locates the second most bugs in the first 1000 lines. But we can also see that the differences to the other metrics in the first group are quite negligible. For our experiments, we applied the commonly used parameter $* = 2$. Fig. 7b shows the absolute bug localization performance in the first 1000 ranking elements for the DEFECTS4J benchmark using the D* metric.

For the ASPECTJ benchmark, however, D* is one of the worst performing metrics, locating no bugs at all within the first 100 lines and only locating 2 bugs within the first 1000 lines. This may be due to the ASPECTJ benchmark not being very suitable for SBFL in general, as we can see that multiple well-known and well-performing SBFL metrics are performing very poorly. Fig. 8b shows the bug localization performance of the D* metric for the ASPECTJ benchmark within the first 10,000 lines.

GP13

GP13 is the result of synthesizing an SBFL metric using an evolutionary algorithm. It was shown to be similarly effective to Op2 by Yoo[12]. Fig. 7c shows the absolute bug localization performance in the first 1000 ranking elements for the DEFECTS4J benchmark using the GP13 metric.

GP13 also performs comparable to Op2 for the ASPECTJ benchmark, being not able to locate a single bug within the first 100 ranked elements and only being able to locate 3 bugs within the first 1,000 ranked elements. Fig. 8c shows the bug localization performance of the GP13 metric for the ASPECTJ benchmark within the first 10,000 lines.

Hyperbolic

Recently, Neelofar et al.[31] proposed to use parameterized hyperbolic functions to localize bugs. It was shown to be at least equally as effective as the other examined SBFL metrics, including Op2 and GP13, among others. The parameters, i.e., K_1, K_2 and K_3 were determined using an evolutionary algorithm or simulated annealing on a training set of bugs, using ten-fold cross validation.

Since the study[31] proposes no concrete values for the parameters, we decided to use the coefficients $K_1 = 0.375$, $K_2 = 0.768$ and $K_3 = 0.711$ which the authors supplied us with. The coefficients were taken from results of a ten-fold cross validation done on the Siemens Test Suite benchmark, as we were told. To avoid any bias, we simply use the coefficients that were generated for the first training set. Note that these coefficients may not be optimal, since they were found using an entirely different benchmark. Though, the plots in Fig. 3, Fig. 5b and Fig. 5d show that the Hyperbolic metric is still among the group of metrics that perform best for the DEFECTS4J benchmark. Fig. 7d shows the absolute bug localization performance in the first 1,000 ranking elements for the DEFECTS4J benchmark using the Hyperbolic metric with the aforementioned coefficients.

For the ASPECTJ benchmark, the Hyperbolic metric performs comparable to both Op2 and GP13, locating no bugs within the first 100 ranked elements and only locating 3 bugs within the first 1,000 ranked elements. Fig. 8d shows the bug localization performance of the Hyperbolic metric for the ASPECTJ benchmark within the first 10,000 lines.

We decided to additionally replicate the methodology used by Neelofar et al.[31] on the DEFECTS4J benchmark, and we performed a ten-fold cross validation using an evolutionary algorithm to determine appropriate coefficients K_1, K_2 and K_3. We randomly distributed the entities in the DEFECTS4J benchmark into 10 sets from which we use one set in each run as the *training set* to determine the coefficients. We then used these coefficients for the Hyperbolic metric and applied it to the respective *test set*, i.e., the set containing all entities in the remaining nine sets. The results are shown in Table 4. For each examined metric and for the Hyperbolic metric with the respective shown coefficients, we compute the median average rankings of bugs in the respective test sets. The table shows that the Hyperbolic metric is performing similarly good to the other better metrics, but it is also never able to actually perform best for any of the ten test sets.

All four additionally examined metrics perform comparably well for both benchmarks. They perform very good on the DEFECTS4J benchmark with D* being the overall best performing metric, and they perform badly on the ASPECTJ benchmark, with D* being one of the worst metrics. We could not exactly replicate the very good performance of the Hyperbolic metric found by Neelofar et al.[31], although it is still among the best performing SBFL metrics for the DEFECTS4J benchmark in our experiments. Same as the Op2 and GP13 metrics that performed quite similarly to each other.

5.2 | RQ$_2$:What is the uncertainty in the assigned suspiciousness scores?

As in our previous studies[27], we use the Area Between Curves (ABC) metric (see Section 3.3.1) to evaluate the *decidedness* of SBFL metrics. In Fig. 2, 6, 7, 8 and Fig. 9, the area between the two curves is highlighted. The smaller this area (i.e., the ABC metric) is, the less random a ranking is and, thus, more reliable for the developer. A value of 100% would represent a completely random ranking (i.e., low decidedness; all ranked elements have the same suspiciousness), whereas a value close to 0% (i.e., high decidedness; all bugs have unique suspiciousness) represents a deterministic ranking.

5.2.1 | Impact of Different Ranking Metrics

Especially in the top ranks, SBFL metrics should rank elements with high confidence and decidedness to be trustworthy for developers. We therefore concentrate only on the first 1,000 lines of the rankings in our experiments. In Fig. 9, we show the absolute bug localization performance of the Tarantula and the Overlap metric for the ASPECTJ and DEFECTS4J benchmarks within the first 1,000 ranked elements. We can see that the Overlap metric is more indecisive for both benchmarks than the Tarantula metric. The ABC value for the DEFECTS4J benchmark is 46.36% for the Overlap metric and only 4.45% for the Tarantula metric. For the ASPECTJ benchmark, the ABC value for the Overlap metric is 55.09% and 14.04% for the Tarantula metric. Note that the ABC metric values can only reasonably be compared within the same benchmark. Comparisons between the two benchmarks can not really be made, since the the ABC metric is tied to the amount and the type of bugs found in the specified range of ranked elements.

Examining the growth of the best- and worst-case rankings for the first 1,000 lines of code for the Overlap ranking metric in Fig. 9a and Fig. 9b, it is evident that the guidance for a developer is not very clear for both examined benchmarks. For ASPECTJ, a developer may find up to $\sim$ 17% of the bugs after examining 1,000 lines of code, or (s)he might find only $\sim$ 2%

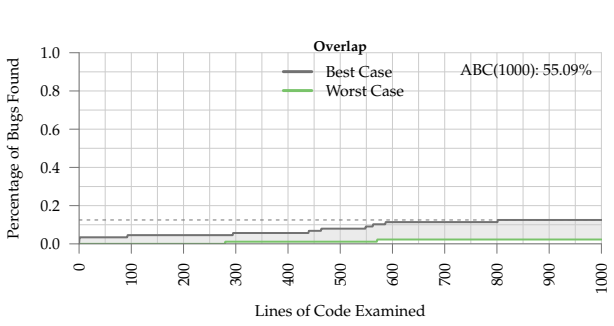
TABLE 4 Results of ten-fold cross validation with hyperbolic function coefficients determined by an evolutionary algorithm on the respective training sets.

SBFL metric	median avg bug localization ranks (absolute wasted effort) for each of the 10 test sets									
Ample	75.5	73	73	74.75	79	79	75.5	88	83	73
Anderberg	47.75	46	**45.75**	47.75	46	50	41	56.5	43.25	39.75
ArithmeticMean	50.25	50.25	49.75	50.75	50	53	49.5	58	49.5	45.75
Cohen	50.25	50.25	49.75	50.75	50	53	49.5	58.5	49.5	45.75
Dice	49.5	47.75	**45.75**	49.5	46	50	41	56.5	43.25	39.75
Euclid	15133.75	14505.75	15309.25	15253.5	15189.5	14845.5	15572	14845.5	15500.5	14961.75
Fleiss	137.5	94.5	91.75	107.5	93.5	134.5	104.5	136.5	104.75	91.75
GeometricMean	49.5	49	49.5	49.75	49.5	51	46	52	47.75	45.75
Goodman	49.5	47.75	**45.75**	49.5	46	50	41	56.5	43.25	39.75
Hamann	15133.75	14505.75	15309.25	15253.5	15189.5	14845.5	15572	14845.5	15500.5	14961.75
Hamming	15133.75	14505.75	15309.25	15253.5	15189.5	14845.5	15572	14845.5	15500.5	14961.75
HarmonicMean	49.5	45.75	49.5	47.75	49.5	50.5	49.5	52	49.5	44.75
Jaccard	49.5	47.75	**45.75**	49.5	46	50	41	56.5	43.25	39.75
Kulczynski1	47.75	46	**45.75**	47.75	46	50	41	56.5	43.25	39.75
Kulczynski2	49.25	45.75	49.25	46	46	50	49	52	46	42.25
M1	15133.75	14505.75	15309.25	15253.5	15189.5	14845.5	15572	14845.5	15500.5	14961.75
M2	48.5	49.5	50.25	49.5	50	51	50	50.5	50	47.5
Ochiai	47.25	46	**45.75**	47.25	46	**48.5**	**40.5**	51	**42.75**	39.75
Ochiai2	49.75	49.75	49.5	50	49.5	51	48.5	52	49.5	45.75
Overlap	308.5	251.5	293.5	292.5	285	285	300	306	293	243.75
RogersTanimoto	15133.75	14505.75	15309.25	15253.5	15189.5	14845.5	15572	14845.5	15500.5	14961.75
Rogot1	142	97.5	95.5	106.75	95.5	136.5	109.5	143	110.75	95.5
Rogot2	49.5	**42.25**	46	**45.75**	46	49.5	46	50.5	46	40.25
RussellRao	321.75	264.5	308.5	308.5	306	301	308	308	307.5	264.5
Scott	142	97.5	95.5	106.75	95.5	136.5	109.5	143	110.75	95.5
SimpleMatching	15133.75	14505.75	15309.25	15253.5	15189.5	14845.5	15572	14845.5	15500.5	14961.75
Sokal	15133.75	14505.75	15309.25	15253.5	15189.5	14845.5	15572	14845.5	15500.5	14961.75
SorensenDice	49.5	47.75	**45.75**	49.5	46	50	41	56.5	43.25	39.75
Tarantula	49.5	47.5	**45.75**	49.25	46	49.5	**40.5**	54	**42.75**	**39.25**
Wong1	321.75	264.5	308.5	308.5	306	301	308	308	307.5	264.5
Wong2	15133.75	14505.75	15309.25	15253.5	15189.5	14845.5	15572	14845.5	15500.5	14961.75
Wong3	10174.75	10137.75	10429.5	10349.5	9843	10407.5	10291.5	10291.5	10741.5	10285
Zoltar	47.25	44.75	49.25	47.5	**45.5**	49.5	49	50.5	45.75	42.25
Op2	49.5	49.75	50.25	49.75	50	51	50	50.5	50	49.25
DStar	**45.75**	43.25	46	**45.75**	**45.5**	**48.5**	**40.5**	**50**	43.25	40
GP13	49.25	49.5	50.25	49.75	50	51	50	50.5	50	48
Hyperbolic	48.75	50	51.5	50	50.5	53.5	53	57	50.5	68.75
K_1	8.093	1.726	3.928	0.253	15.630	26.324	16.009	18.856	6.688	5.112
K_2	20.008	4.423	16.963	3.521	36.928	69.183	25.115	56.850	17.206	54.097
K_3	1.368	0.243	0.479	0.361	1.839	1.845	1.508	1.704	1.089	0.094

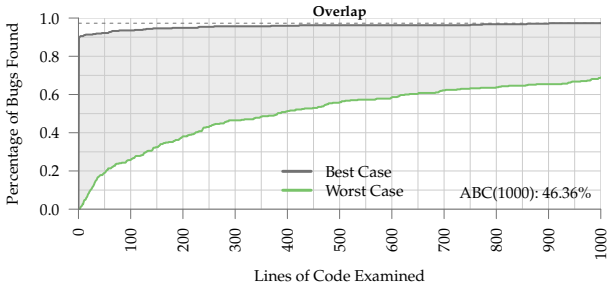
in the worst case. For DEFECTS4J, a developer is theoretically able to find ∼90% of the bugs within the first few lines if (s)he is very lucky, while in the worst case, (s)he would not find more than ∼70% of the bugs after examining the first 1,000 lines. Overlap has a very high ABC value in the first 1,000 lines of code for both examined benchmarks.

Looking at the plots for the Tarantula metric in Fig. 9c and Fig. 9d, we see that, for ASPECTJ, a developer may also find up to ∼17% of the bugs after examining 1,000 lines of code, but in the worst case, (s)he is still able to find nearly as much bugs as in the best case. For DEFECTS4J, we see that Tarantula is much more decided than the Overlap metric, while still finding more than 90% of the bugs within the first 1,000 lines in the worst case.

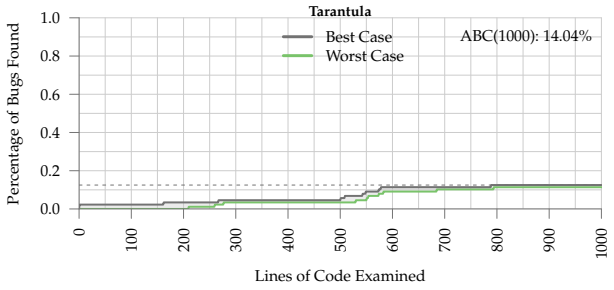
As found in our previous studies[27], we confirm the result that if multiple elements share the same suspiciousness, SBFL reaches its limitations. We also see that if the conditions for SBFL are better – as in the case of the DEFECTS4J benchmark, the best performing SBFL metrics usually also have lower ABC values (cf. Section 5.5). The decreased performance in the



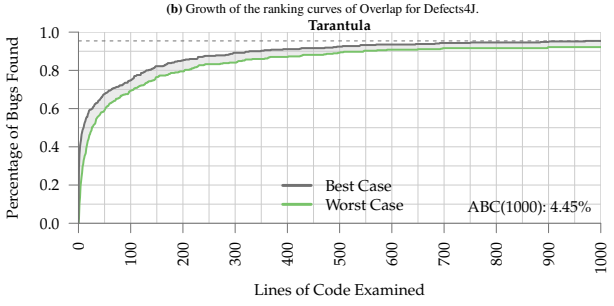
(a) Growth of the ranking curves of Overlap for AspectJ.



(b) Growth of the ranking curves of Overlap for Defects4J.



(c) Growth of the ranking curves of Tarantula for AspectJ.



(d) Growth of the ranking curves of Tarantula for Defects4J.

FIGURE 9 The fault localization decidedness in the first 1,000 lines of examined code.

ASPECTJ benchmark can be explained by the noise that is often introduced by the existence of failing test cases in the test suites that are not at all related to the bugs under consideration.

We confirm the results found in our previous study[27]. Though, we see a general trend that the metrics that perform better on the DEFECTS4J benchmark also have a lower ABC metric value than metrics that perform worse.

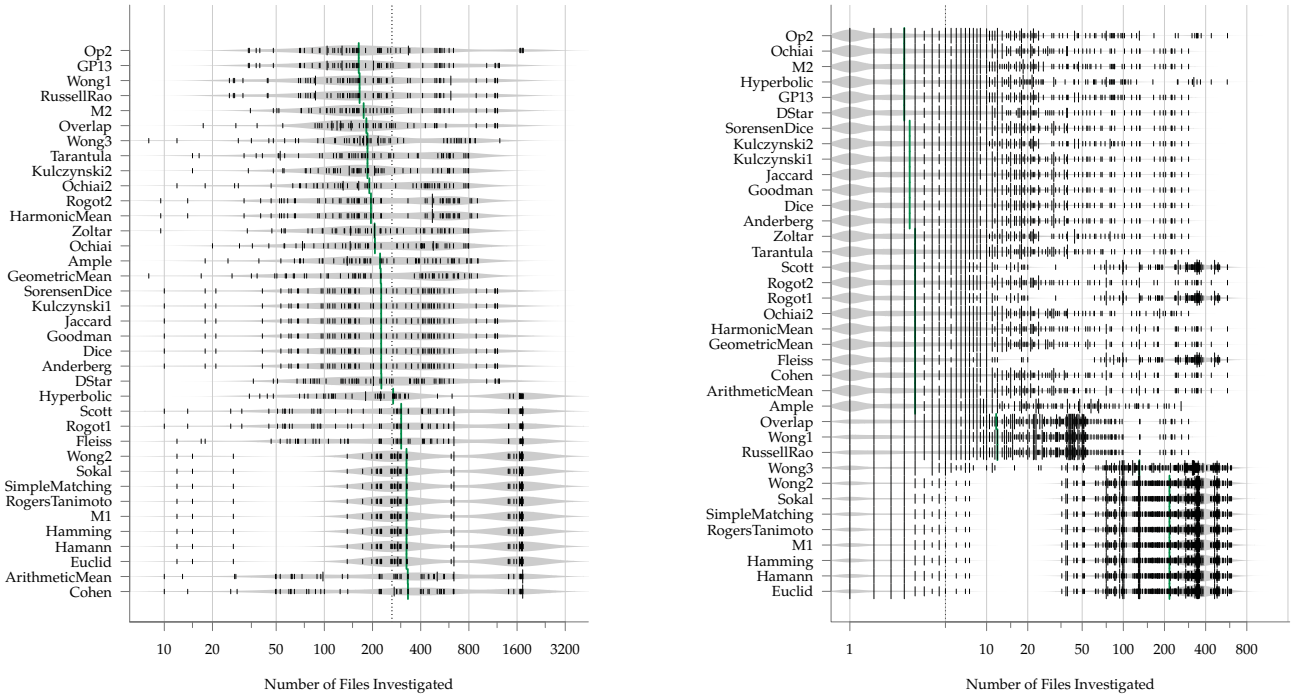
5.3 | RQ_3: What is the number of files inspected when following SBFL techniques?

The Number of Files Investigated (NFI) metric (cf. Section 3.3.2) counts the files a developer has to look at until (s)he finds the prominent bug.

As done in our previous studies[27], Fig. 10 plots the average number of files a developer has to investigate prior to finding the file containing the prominent bug for all buggy versions contained in the respective benchmarks. In Fig. 10, the vertical axes contain entries for each of the examined 37 metrics, and the horizontal axes represent the number of files a developer has to investigate, plotted on a logarithmic scale. The minor ticks mark the arithmetic mean between the next lower and higher major ticks. Each line in the plot represents the average number of files that need to be investigated for a specific bug using the respective SBFL ranking metric, and a longer line represents multiple bugs being found within the same number of files. The thick green line represents the median of the number of investigated files for all bugs examined for each SBFL metric, while the thin dotted line in the background marks the median of the medians of all examined ranking metrics. The density of localized bugs is represented by the shaded area, where larger shaded areas implicate a larger number of prominent bugs found within the respective range. Sorting of SBFL metrics is done based on their medians in ascending order.

The first general result is that the number of files that have to be touched is much smaller for the DEFECTS4J benchmark than for the ASPECTJ benchmark. Half of the examined bugs in the DEFECTS4J benchmark can be found by looking at less than 5 files using most of the SBFL metrics. The best metrics have a median *nfi* value of 2.5. Multiple bugs are even found within the first examined file for each of the examined SBFL metrics, while a developer has to investigate more than 10 files on average for the ASPECTJ benchmark. In order to find 50% of the bugs, a developer even has to investigate more than 150 files on average.

We see that both Op2 and GP13 perform better for both benchmarks in terms of the NFI metric. For the DEFECTS4J benchmark, the Hyperbolic and D^* metrics also perform well, while they perform worse for the ASPECTJ benchmark.



(a) NFI for AspectJ.

(b) NFI for Defects4J.

FIGURE 10 Average number of files investigated in order to find all prominent bugs.

While the number of files that has to be investigated is much smaller for the DEFECTS4J benchmark, for some bugs, a developer would still have to look at ~ 500 files, independently from the used metric, which is still infeasible. The additional four SBFL metrics that are examined in this study are also not able to improve over the other metrics. This is another implication that the information used by pure SBFL metrics is not enough to improve the bug localization performance beyond a certain point.

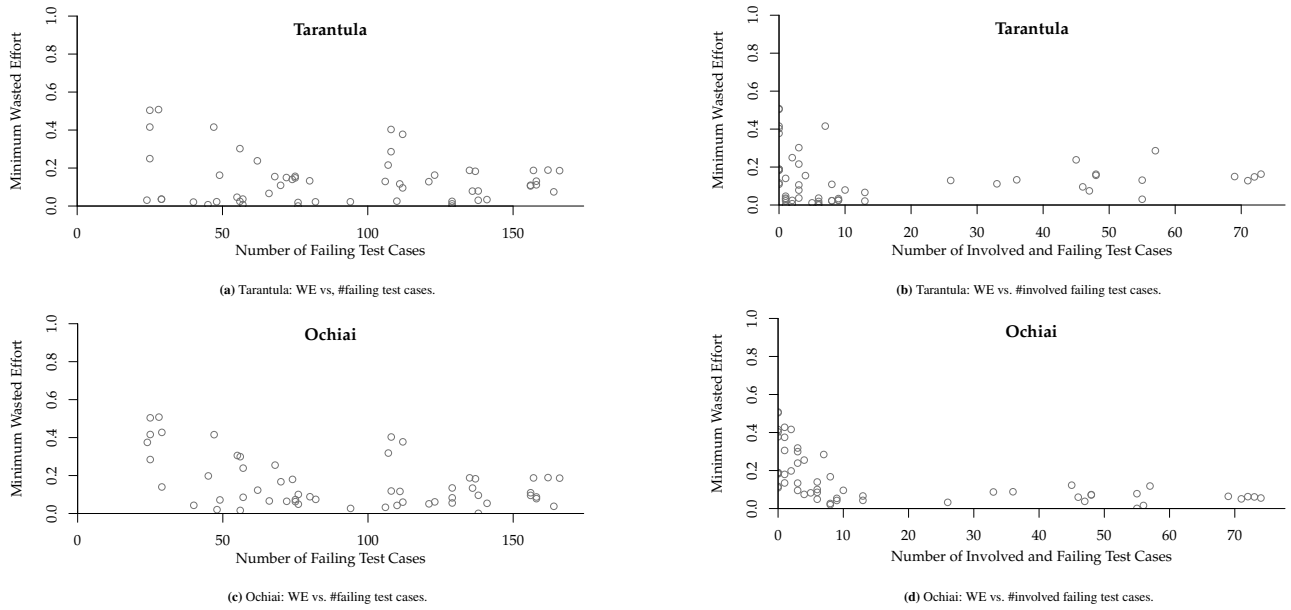
50% of the bugs in the DEFECTS4J benchmark can be localized within less than 5 files for most of the examined SBFL metrics. The additional SBFL metrics are not able to significantly improve the results for the ASPECTJ benchmark.

5.4 | RQ_4: What is the relation between the number of related and unrelated failing test cases and SBFL's accuracy?

We answer RQ_4 only based on the ASPECTJ benchmark, since we are interested in the effect of realistic test suites with related and unrelated failing test cases. An evaluation on the DEFECTS4J benchmark would not present a realistic picture, since the benchmark is curated in a way that each DEFECTS4J bug is exposed by a limited number of directly related failing tests. As a result, we cannot see the effect of unrelated failing test cases.

Fig. 11 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 Fig. 11a and 11c reveals that there is no particular relation between the two variables. This can be explained with the same argument as provided in Section 5.1 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 Fig. 11b and 11d) 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 (Fig. 11a-11d), 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[8], 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 (please compare[8]) and some high p-values for the smaller correlation coefficients.



(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.

FIGURE 11 The relation between the number of [involved] failing test cases and SBFL's accuracy.

The second data set considers only involved failing test cases (Fig. 11b and 11d) 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.

5.5 | Effectiveness Metrics in Comparison

Table 5 shows selected effectiveness metrics (cf. Section 3) for all examined SBFL ranking metrics for both examined benchmarks. All columns but the last column are percentage values rounded to two digits. If existing, the best value of each effectiveness metric is printed in bold font.

For the $\{min, max\}\_we_{mean}$ metrics, the wasted effort metric of all bugs has been aggregated using the arithmetic mean. The $\{min, max\}\_pbl_{\{10,1000\}}$ metrics show the minimum and maximum percentage of located bugs within the first 10 or $1,000$ ranked elements for each examined SBFL metric. We chose to show the values for the first 10 ranked elements, since this is the most important range for human developers, while the first $1,000$ ranked elements may still be relevant to automated program repair tools, for example. $ABC(1000)$ (Eq. 19) refers to the Area Between Two Curves within the first $1,000$ ranked elements as defined in Section 3.3.1. The last column shows the median number of files investigated (Section 3.3.2) metric. The last row of the table contains the arithmetic mean of each effectiveness metric for all ranking metrics.

We see that employing the additional examined metrics is not able to significantly improve the performance of SBFL on the ASPECTJ benchmark. As we have seen in Section 5.1, the additionally examined metrics even tend to perform worse on the ASPECTJ benchmark. This implies again that the ASPECTJ benchmark does not offer suitable conditions for SBFL to function satisfyingly for a developer. With a benchmark that is much more suited for SBFL, like DEFECTS4J in this case, we are able to confirm the strength of the additionally examined SBFL metrics.

SBFL techniques work significantly better on the DEFECTS4J benchmark than on the ASPECTJ benchmark. Still, only $\sim 30\%$ of the bugs can be found reliably within the first 10 ranked elements.

TABLE 5 Various SBFL effectiveness metrics for the different ranking metrics for the ASPECTJ benchmark (left) and the DEFECTS4J benchmark (right).

Ranking metric	min_ue_{mean}	max_ue_{mean}	min_pbl_{10}	max_pbl_{10}	min_pbl_{1000}	max_pbl_{1000}	$ABC(1000)$	$median\_nfi$
Ample	22.78	47.42	0.00	0.00	4.55	4.55	19.83	390.75
Anderberg	17.83	55.21	0.00	0.00	5.68	6.82	6.06	420.00
ArithmeticMean	26.02	55.25	0.00	0.00	10.23	10.23	2.21	506.25
Cohen	25.97	55.20	0.00	0.00	10.23	10.23	2.21	528.75
Dice	17.83	55.21	0.00	0.00	5.68	6.82	6.06	420.00
Euclid	36.10	66.55	0.00	0.00	3.41	**21.59**	51.81	841.75
Fleiss	26.89	56.02	0.00	0.00	9.09	9.09	2.45	531.50
GeometricMean	20.67	**45.36**	0.00	0.00	**11.36**	11.36	2.67	462.25
Goodman	17.83	55.21	0.00	0.00	5.68	6.82	6.06	420.00
Hamann	36.10	66.55	0.00	0.00	3.41	**21.59**	51.81	841.75
Hamming	36.10	66.55	0.00	0.00	3.41	**21.59**	51.81	841.75
HarmonicMean	20.83	45.51	0.00	0.00	**11.36**	11.36	2.67	473.00
Jaccard	17.83	55.21	0.00	0.00	5.68	6.82	6.06	420.00
Kulczynski1	17.83	55.21	0.00	0.00	5.68	6.82	6.06	420.00
Kulczynski2	21.44	47.50	0.00	0.00	4.55	4.55	17.30	**238.50**
M1	36.10	66.55	0.00	0.00	3.41	**21.59**	51.81	841.75
M2	19.00	56.37	0.00	0.00	2.27	2.27	34.00	254.50
Ochiai	21.36	47.43	0.00	0.00	5.68	5.68	13.54	404.75
Ochiai2	20.56	46.67	0.00	0.00	6.82	9.09	2.59	430.50
Overlap	**14.43**	59.68	0.00	**3.41**	2.27	12.50	55.09	258.75
RogersTanimoto	36.10	66.55	0.00	0.00	3.41	**21.59**	51.81	841.75
Rogot1	25.98	55.11	0.00	0.00	10.23	11.36	**2.19**	506.50
Rogot2	20.83	45.51	0.00	0.00	**11.36**	11.36	2.67	473.00
RussellRao	18.50	56.99	0.00	1.14	0.00	10.23	84.81	246.25
Scott	25.98	55.11	0.00	0.00	10.23	11.36	**2.19**	506.50
SimpleMatching	36.10	66.55	0.00	0.00	3.41	**21.59**	51.81	841.75
Sokal	36.10	66.55	0.00	0.00	3.41	**21.59**	51.81	841.75
SorensenDice	17.83	55.21	0.00	0.00	5.68	6.82	6.06	420.00
Tarantula	19.87	45.93	0.00	2.27	**11.36**	12.50	14.04	358.00
Wong1	18.50	56.99	0.00	1.14	0.00	10.23	84.81	246.25
Wong2	36.10	66.55	0.00	0.00	3.41	**21.59**	51.81	841.75
Wong3	24.14	50.66	0.00	0.00	2.27	4.55	20.20	246.25
Zoltar	21.40	47.46	0.00	0.00	3.41	4.55	19.26	343.50
Op2	25.93	52.61	0.00	0.00	3.53	3.53	26.40	338.50
DStar	19.23	56.81	0.00	0.00	2.35	2.35	31.05	408.50
GP13	19.27	56.83	0.00	0.00	3.53	3.53	26.40	338.50
Hyperbolic	35.29	62.84	0.00	0.00	3.53	3.53	26.40	868.75
Mean value	24.61	55.92	0.0	0.22	5.45	10.64	25.56	503.08

Ranking metric	min_ue_{mean}	max_ue_{mean}	min_pbl_{10}	max_pbl_{10}	min_pbl_{1000}	max_pbl_{1000}	$ABC(1000)$	$median\_nfi$
Ample	1.95	**3.27**	27.03	41.89	88.65	90.00	3.24	3.00
Anderberg	0.64	5.20	32.16	49.73	91.89	95.14	4.36	2.75
ArithmeticMean	4.29	6.13	32.16	49.19	90.27	92.43	3.42	3.00
Cohen	4.30	6.16	31.89	49.46	90.27	92.16	3.28	3.00
Dice	0.65	5.20	32.16	49.73	91.89	95.14	4.35	2.75
Euclid	54.98	64.70	10.54	25.14	20.27	31.08	36.78	217.50
Fleiss	24.33	26.16	31.08	46.76	70.27	72.16	3.91	3.00
GeometricMean	4.29	6.12	32.43	49.46	90.27	92.16	3.28	3.00
Goodman	0.65	5.20	32.16	49.73	91.89	95.14	4.35	2.75
Hamann	54.98	64.70	10.54	25.14	20.27	31.08	36.78	217.50
Hamming	54.98	64.70	10.54	25.14	20.27	31.08	36.78	217.50
HarmonicMean	4.28	6.11	31.89	49.19	90.27	92.16	3.29	3.00
Jaccard	0.65	5.20	32.16	49.73	91.89	95.14	4.35	2.75
Kulczynski1	0.65	5.20	32.16	49.73	91.89	95.14	4.35	2.75
Kulczynski2	0.75	5.31	31.08	48.11	91.08	94.32	4.52	2.75
M1	54.98	64.70	10.54	25.14	20.27	31.08	36.78	217.50
M2	0.78	5.35	27.84	48.92	91.08	94.59	4.75	**2.50**
Ochiai	0.60	5.16	32.43	49.73	**92.16**	95.41	4.48	**2.50**
Ochiai2	0.65	6.78	32.43	49.73	90.54	95.14	5.52	3.00
Overlap	**0.34**	8.15	2.70	**90.54**	68.65	**97.30**	46.36	11.75
RogersTanimoto	54.98	64.70	10.54	25.14	20.27	31.08	36.78	217.50
Rogot1	23.95	25.53	30.27	45.95	70.81	72.43	3.63	3.00
Rogot2	4.78	5.36	32.43	48.92	91.08	91.89	2.14	3.00
RussellRao	0.44	8.16	2.70	87.30	68.65	96.49	45.14	12.00
Scott	23.95	25.53	30.27	45.95	70.81	72.43	3.63	3.00
SimpleMatching	54.98	64.70	10.54	25.14	20.27	31.08	36.78	217.50
Sokal	54.98	64.70	10.54	25.14	20.27	31.08	36.78	217.50
SorensenDice	0.65	5.20	32.16	49.73	91.89	95.14	4.35	2.75
Tarantula	0.65	5.22	30.81	49.46	**92.16**	95.41	4.45	3.00
Wong1	0.44	8.16	2.70	87.30	68.65	96.49	45.14	12.00
Wong2	54.98	64.70	10.54	25.14	20.27	31.08	36.78	217.50
Wong3	41.59	50.07	12.70	26.22	31.62	41.62	24.85	131.00
Zoltar	0.73	5.30	30.54	47.57	91.35	94.59	4.45	3.00
Op2	4.22	4.54	30.00	47.57	90.54	91.08	**2.11**	**2.50**
DStar	0.59	5.15	**32.70**	49.19	**92.16**	95.41	4.48	**2.50**
GP13	0.82	5.38	30.00	47.84	90.54	93.78	4.40	**2.50**
Hyperbolic	3.16	5.28	29.46	47.57	90.54	91.35	3.00	**2.50**
Mean value	16.1	21.28	23.91	46.04	70.7	77.3	14.97	53.43

5.6 | Threats to Validity

The original study[27] had 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. Specifically, there was a threat to external validity that the results obtained from the ASPECTJ benchmark cannot be generalized to other large scale software systems. In this paper, we mitigate the threat by including another benchmark, namely DEFECTS4J, into the experiments. By cross analysis of the results, we could identify where the benchmarks are similar and where the results diverge. As a result, we provide a stronger confidence in the generalizability of the results.

A second potential threat to validity of the original study[27] was in the manual classification of the bug locations by the authors. This classification was done very conservative, and only bugs were considered where the authors had high confidence. Furthermore, all arguments supporting the classification were documented and we made the commented code versions for the

bug set of ASPECTJ available at[44] to be checked for future investigations. By adding DEFECTS4J as a second benchmark, we also mitigated the risk of a bias classification, since the bugs were collected and classified by the curators of the DEFECTS4J benchmark[26].

6 | CONCLUSIONS & LESSONS LEARNED

6.1 | Main conclusion

Pure 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[7]. Several SBFL approaches have been evaluated only on small and medium-size benchmarks and under the unrealistic assumption of a developer sequentially examining elements in the ranked list of suspects[29].

In this paper, we empirically evaluated 37 pure SBFL approaches, including Naish2 (Op2)[8], D*[30], the GP-evolved ranking metric GP13[12], and the family of SBFL metrics based on hyperbolic functions[31] for the debugging of a large-scale project. For this empirical evaluation, we used two large-scale benchmarks, namely ASPECTJ via the iBugs[28] repository and DEFECTS4J[26]. To compare the different SBFL approaches, we used both a set of established metrics (e.g., the absolute *we* and *Hit@X*) and two additional ones that we defined, namely the number of files investigated (*nfi*) as well as the area between curves (*ABC*) metric, which try to better approximate more realistic assumptions about developers' behavior.

Even with the additionally examined SBFL metrics, at most 11 out of 88 bugs can be found after examining the 1000 top ranking suspicious lines for the ASPECTJ benchmark, requiring on average an inspection of about 250 files to discover any bug. These results are identical with our previous study[27]. For the DEFECTS4J benchmark, at most 353 out of 370 bugs can be found within the first 1000 top ranking suspicious lines with many of the examined metrics requiring the investigation of less than 5 files, on average, to discover a bug. If we only look at the first 10 suspicious lines in the rankings, which is more realistic for a human developer, only $\sim 40\%$ and $\sim 33\%$ of the bugs can be localized in the average case and the worst case, respectively. In order to reliably localize 90% of the bugs with D*, ~ 450 lines have to be inspected by the developer in the worst case. These results provide an indication that pure SBFL approaches are not suitable for human developers.

6.2 | Additional Lessons Learned

In addition to the main conclusions of the paper, we learn the following additional lessons from the empirical evaluation:

Lesson 1. The performance of pure SBFL metrics heavily depends on the selected program used for investigation.

Lesson 2. Many of the examined SBFL metrics performed similarly well on the DEFECTS4J benchmark, and the practical impact of using one of the metrics over another is not significant for a human developer.

Lesson 3. We confirm our results[27] that the number of lines and files that need to be inspected based on SBFL rankings can be reduced by (higher quality) test cases that execute the bug.

Lesson 4. Using a more tailored benchmark like DEFECTS4J, the randomness in the SBFL rankings decreases overall, and better performing SBFL metrics generally contain less randomness.

6.3 | Beyond Pure SBFL

Our results highlight that pure SBFL – which computes suspicious scores from raw program spectra collected by running a test suite, is not fully effective in identifying faulty code. It would be interesting to explore hybrid and extended SBFL methods, as well as other related fault localization (FL) methods which may hold more promise. In the literature, several directions beyond pure SBFL have been explored:

1. Several past studies have merged or replaced SBFL with other dynamic analysis techniques. For example, Alves et al.[45], Mao et al.[46], and Wen[47] have combined SBFL with dynamic program slicing. Dynamic program slicing is used to exclude uninteresting program elements; the remaining program elements are then ranked using SBFL. Several studies employ machine learning to analyze execution traces: Le et al.[48] performed dynamic invariant mining and use the mined invariants to differentiate faulty and non-faulty program elements using a learning to rank method; Cellier et al.[49] employed both association rule mining and formal concept analysis to identify faulty program elements from a collection of correct and

faulty execution traces; Zhang et al.[50] boosted SBFL by employing an approach based on the PageRank algorithm to differentiate tests based on their likely importance; Wang et al.[51], Lucia et al.[52] and Xuan et al.[39] combined multiple SBFL solutions using a genetic algorithm, data fusion methods, and a classification algorithm, respectively. The idea of combining SBFL metrics is also being empirically evaluated by Zou et al.[53].

2. A different direction is the composition of static and dynamic analysis for fault localization. Two works in this direction include those by Jiang et al.[54] and Zhao et al.[55] that statically extract control flow graphs and use it to improve SBFL. The work by Feyzi and Parsa[56] uses a static fault-proneness analysis. De Souza et al.[57] use Code Hierarchy (CH) and Integration Coverage-based Debugging (ICD) techniques to provide better ranking of methods to be inspected. The approach by Sohn and Yoo[58] combines code and change metrics with SBFL techniques. Another work by Ren and Ryder[59] identifies methods that are likely to cause a test failure by analyzing a statically constructed call graph for the failed test, and counting the number of ancestors and descendants of methods in the graph. Finally, the approach of Neelofar et al.[60] combines SBFL with a static analysis techniques which categorizes individual program statements, for example, into control statements, assignment statements, return statements, etc.

3. Another direction is the composition of dynamic and text analysis for effective FL. For example, Le et al.[61] merged information retrieval (IR) based fault localization (IRFL) and SBFL for a more effective fault localisation. Their proposed solution analyzes two types of debugging hints: (1) test cases and their outcomes, along with (2) textual contents in associated bug reports.

4. Yet another direction is mutation based fault localization[62,63,24,64,65,66,67]. Program mutants are generated and used to detect faulty program elements. The intuition is that mutants killed by failing test cases are likely to indicate locations of the faults.

5. A final direction is to use SBFL in an interactive, feedback-driven process similar to the work on interactive[68,69], feedback-based[70] or whyline debugging[71]. Here we have to mention the work on Enlightened Debugging[72] which, based on the SBFL results, selects failing test cases and identifies spurious method invocations. For these method invocations, the approach generates a debugging query based on the methods' input-output relations and consults the user. The user inspects the input-output relations and acts as an oracle for correctness of the method invocations. As a result, the user guides a search process to better localize faults.

Future research may want to assess effectiveness of these methods more comprehensively (following a similar methodology that is done in this work) as well as extend or combine them to achieve a high-enough accuracy for fault localization (FL) to be adopted in practice – c.f.,[40,73].

Another promising direction is to improve the quality of test cases that are used as input to SBFL techniques. A number of specialized test (case/suite) generation[74,75], prioritization[76,77], reduction[78,79], purification and cleaning[80,81], and assessment[74,82] methods have been proposed in the literature to improve fault localization. One difficulty in generating new test cases for improved SBFL is the unavailability of test oracles. Interestingly, several studies have proposed methods that can perform fault localization without the need of test case oracles – c.f.,[21,83].

References

1. Orso Alessandro, Rothermel Gregg. Software Testing: A Research Travelogue (2000–2014). In: Proceedings of the on Future of Software Engineering:117–132; 2014; New York, NY, USA.

2. Böhme Marcel, Soremekun Ezekiel O., Chattopadhyay Sudipta, Ugherughe Emamurho, Zeller Andreas. Where is the bug and how is it fixed? an experiment with practitioners. In: Bodden Eric, Schäfer Wilhelm, Deursen Arie, Zisman Andrea, eds. *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, :117–128; 2017.

3. Cleve Holger, Zeller Andreas. Locating Causes of Program Failures. In: Proceedings of the 27th International Conference on Software Engineering:342–351; 2005; New York, NY, USA.

4. Jones James A., Harrold Mary Jean, Stasko John. Visualization of Test Information to Assist Fault Localization. In: Proceedings of the 24th International Conference on Software Engineering:467–477; 2002; New York, NY, USA.

5. Liblit Ben, Naik Mayur, Zheng Alice X., Aiken Alex, Jordan Michael I.. Scalable Statistical Bug Isolation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation:15–26; 2005.

6. Liu Chao, Fei Long, Yan X., Han Jiawei, Midkiff S.P.. Statistical Debugging: A Hypothesis Testing-Based Approach. *IEEE Transactions on Software Engineering.* 2006;32(10):831–848.

7. Wong W. Eric, Gao Ruizhi, Li Yihao, Abreu Rui, Wotawa Franz. A Survey on Software Fault Localization. *IEEE Trans. Software Eng.*. 2016;42(8):707–740.

8. Naish Lee, Lee Hua Jie, Ramamohanarao Kotagiri. A Model for Spectra-based Software Diagnosis. *ACM Trans. Softw. Eng. Methodol.*. 2011;20(3):1–32.

9. Xie Xiaoyuan, Chen Tsong Yueh, Kuo Fei-Ching, Xu Baowen. A Theoretical Analysis of the Risk Evaluation Formulas for Spectrum-based Fault Localization. *ACM Trans. Softw. Eng. Methodol.*. 2013;22(4):1–40.

10. Xie Xiaoyuan, Kuo Fei-Ching, Chen Tsong Yueh, Yoo Shin, Harman Mark. Provably Optimal and Human-Competitive Results in SBSE for Spectrum Based Fault Localisation. In: Search Based Software Engineering - 5th International Symposium, SSBSE 2013, vol. 8084: :224–238; 2013.

11. Yoo Shin, Xie Xiaoyuan, Kuo Fei-Ching, Chen Tsong Yueh, Harman Mark. Human Competitiveness of Genetic Programming in Spectrum-Based Fault Localisation: Theoretical and Empirical Analysis. *ACM Trans. Softw. Eng. Methodol.*. 2017;26(1):4:1–4:30.

12. Yoo Shin. Evolving Human Competitive Spectra-Based Fault Localisation Techniques. In: Search Based Software Engineering, vol. 7515: :244–258; 2012.

13. Do Hyunsook, Elbaum Sebastian, Rothermel Gregg. Supporting Controlled Experimentation with Testing Techniques: An Infrastructure and Its Potential Impact. *Empirical Softw. Engg.*. 2005;10(4):405–435.

14. Abreu Rui, Zoeteweij Peter, Gemund Arjan J. C. van. An Evaluation of Similarity Coefficients for Software Fault Localization. In: Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing:39–46; 2006; Washington, DC, USA.

15. Abreu Rui, Zoeteweij Peter, Gemund Arjan J. C.. On the Accuracy of Spectrum-based Fault Localization. In: Proceedings of the Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION:89–98; 2007; Washington, DC, USA.

16. Gong Liang, Lo David, Jiang Lingxiao, Zhang Hongyu. Diversity Maximization Speedup for Fault Localization. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering:30–39; 2012; New York, NY, USA.

17. Abreu Rui, Zoeteweij Peter, Gemund Arjan J. C. van. Spectrum-Based Multiple Fault Localization. In: Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering:88–99; 2009; Washington, DC, USA.

18. Jones James A., Bowring James F., Harrold Mary Jean. Debugging in Parallel. In: Proceedings of the 2007 International Symposium on Software Testing and Analysis:16–26; 2007; New York, NY, USA.

19. Liu Chao, Yan Xifeng, Fei Long, Han Jiawei, Midkiff Samuel P.. SOBER: Statistical Model-based Bug Localization. In: Proceedings of 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering:286–295; 2005.

20. Lucia , Lo David, Xia Xin. Fusion Fault Localizers. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering:127–138; 2014.

21. Xia Xin, Gong Liang, Le Tien-Duy B., Lo David, Jiang Lingxiao, Zhang Hongyu. Diversity maximization speedup for localizing faults in single-fault and multi-fault programs. *Autom. Softw. Eng.*. 2016;23(1):43–75.

22. Wong W. Eric, Qi Yu, Zhao Lei, Cai Kai-Yuan. Effective Fault Localization Using Code Coverage. In: Proceedings of the 31st Annual International Computer Software and Applications Conference:449–456; 2007.

23. Zhao Lei, Zhang Zhenyu, Wang Lina, Yin Xiaodan. PAFL: Fault Localization via Noise Reduction on Coverage Vector. In: Proceedings of the 23rd International Conference on Software Engineering & Knowledge Engineering:203–206; 2011.

24. Li Xia, Zhang Lingming. Transforming programs and tests in tandem for fault localization. *Proceedings of the ACM on Programming Languages.* 2017;1(OOPSLA):92.

25. Pearson Spencer, Campos José, Just René, et al. Evaluating and improving fault localization techniques. In: Proceedings of the ACM/IEEE International Conference on Software Engineering; 2017.

26. Just René, Jalali Darioush, Ernst Michael D.. 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:437–440; 2014; New York, NY, USA.

27. Keller F., Grunske L., Heiden S., Filieri A., Hoorn A., Lo D.. 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):114–125; 2017.

28. Dallmeier Valentin, Zimmermann Thomas. Extraction of Bug Localization Benchmarks from History. In: Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering:433–436; 2007; New York, NY, USA.

29. Parnin Chris, Orso Alessandro. Are Automated Debugging Techniques Actually Helping Programmers?. In: Proceedings of the 2011 International Symposium on Software Testing and Analysis:199–209; 2011.

30. Wong W Eric, Debroy Vidroha, Li Yihao, Gao Ruizhi. Software fault localization using DStar (D*). In: Software Security and Reliability (SERE), 2012 IEEE Sixth International Conference on:21–30IEEE; 2012.

31. Neelofar N., Naish Lee, Ramamohanarao Kotagiri. Spectral-based fault localization using hyperbolic function. *Software: Practice and Experience.* 2017;:1–24. spe.2527.

32. Reps Thomas, Ball Thomas, Das Manuvir, Larus James. The use of program profiling for software maintenance with applications to the year 2000 problem. In: Software EngineeringâĂŤESEC/FSE'97. 1997 (pp. 432–449).

33. Harrold Mary Jean, Rothermel Gregg, Sayre Kent, Wu Rui, Yi Liu. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability.* 2000;10(3):171–194.

34. Digiuseppe Nicholas, Jones James A.. Fault Density, Fault Types, and Spectra-based Fault Localization. *Empirical Softw. Engg..* 2015;20(4):928–967.

35. Choi Seung-Seok, Cha Sung-Hyuk, Tappert Charles C. A survey of binary similarity and distance measures. *Journal of Systemics, Cybernetics and Informatics.* 2010;8(1):43–48.

36. Naish Lee, Lee Hua Jie. Duals in spectral fault localization. In: Software Engineering Conference (ASWEC), 2013 22nd Australian:51–59IEEE; 2013.

37. Abreu Rui, Zoeteweij Peter, Golsteijn Rob, Gemund Arjan J. C.. A Practical Evaluation of Spectrum-based Fault Localization. *J. Syst. Softw..* 2009;82(11):1780–1792.

38. Casanova Paulo, Schmerl Bradley, Garlan David, Abreu Rui. Architecture-based Run-time Fault Diagnosis. In: Proceedings of the 5th European Conference on Software Architecture:261–277; 2011; Berlin, Heidelberg.

39. Xuan Jifeng, Monperrus Monperrus. Learning to Combine Multiple Ranking Metrics for Fault Localization. In: Proceedings of the International Conference on Software Maintenance and Evolution:191–200; 2014.

40. Kochhar Pavneet Singh, Xia Xin, Lo David, Li Shanping. Practitioners' Expectations on Automated Fault Localization. In: Proceedings of the 25th International Symposium on Software Testing and Analysis:165–176; 2016; New York, NY, USA.

41. AspectJ language extension http://www.eclipse.org/aspectj.

42. Openhub . The AspectJ Open Source Project on Open Hub : Languages Page https://www.openhub.net/p/freshmeat_aspectj/analyses/latest/languages_summary[Online; accessed 1-July-2018]; 2018.

43. Sobreira Victor, Durieux Thomas, Delfim Fernanda Madeiral, Monperrus Martin, Almeida Maia Marcelo. Dissection of a bug dataset: Anatomy of 395 patches from Defects4J. In: Oliveto Rocco, Penta Massimiliano Di, Shepherd David C., eds. *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, :130–140IEEE Computer Society; 2018.

44. Keller Fabian, Grunske Lars, Heiden Simon, Filieri Antonio, Hoorn Andre, Lo David. An Evaluation of Pure Spectrum-Based Fault Localization Techniques for Large-Scale Software Systems - Additional Material www.informatik.hu-berlin.de/de/forschung/gebiete/se/research/material/SPE1819.

45. Alves Elton, Gligoric Milos, Jagannath Vilas, d'Amorim Marcelo. Fault-localization Using Dynamic Slicing and Change Impact Analysis. In: Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering:520–523; 2011; Washington, DC, USA.

46. Mao Xiaoguang, Lei Yan, Dai Ziying, Qi Yuhua, Wang Chengsong. Slice-based statistical fault localization. *Journal of Systems and Software.* 2014;89:51–62.

47. Wen Wanzhi. Software fault localization based on program slicing spectrum. In: 34th International Conference on Software Engineering, ICSE 2012:1511–1514; 2012.

48. Le Tien-Duy B., Lo David, Le Goues Claire, Grunske Lars. A learning-to-rank based fault localization approach using likely invariants. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016:177–188; 2016.

49. Cellier Peggy, Ducassé Mireille, Ferré Sébastien, Ridoux Olivier. Formal Concept Analysis Enhances Fault Localization in Software. In: Proceedings of the 6th International Conference on Formal Concept Analysis:273–288; 2008; Berlin, Heidelberg.

50. Zhang Mengshi, Li Xia, Zhang Lingming, Khurshid Sarfraz. Boosting spectrum-based fault localization using PageRank. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis:261–272; 2017.

51. Wang Shaowei, Lo David, Jiang Lingxiao, Lucia , Lau Hoong Chuin. Search-based fault localization. In: 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011):556–559; 2011.

52. Lucia , Lo David, Xia Xin. Fusion fault localizers. In: ACM/IEEE International Conference on Automated Software Engineering, ASE '14:127–138; 2014.

53. Zou Daming, Liang Jingjing, Xiong Yingfei, Ernst Michael D., Zhang Lu. An Empirical Study of Fault Localization Families and Their Combinations. *CoRR.* 2018;abs/1803.09939.

54. Jiang Lingxiao, Su Zhendong. Context-aware Statistical Debugging: From Bug Predictors to Faulty Control Flow Paths. In: Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering:184–193; 2007; New York, NY, USA.

55. Zhao Lei, Wang Lina, Yin Xiaodan. Context-Aware Fault Localization via Control Flow Analysis. *JSW.* 2011;6(10):1977–1984.

56. Feyzi Farid, Parsa Saeed. FPA-FL: Incorporating static fault-proneness analysis into statistical fault localization. *Journal of Systems and Software.* 2018;136:39–58.

57. Souza Higor Amario, Mutti Danilo, Chaim Marcos Lordello, Kon Fabio. Contextualizing spectrum-based fault localization. *Information & Software Technology.* 2018;94:245–261.

58. Sohn Jeongju, Yoo Shin. FLUCCS: using code and change metrics to improve fault localization. In: Bultan Tevfik, Sen Koushik, eds. *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, :273–283ACM; 2017.

59. Ren Xiaoxia, Ryder Barbara G.. Heuristic ranking of java program edits for fault localization. In: Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2007:239–249; 2007.

60. Neelofar , Naish Lee, Lee Jason, Ramamohanarao Kotagiri. Improving spectral-based fault localization using static analysis. *Softw., Pract. Exper.*. 2017;47(11):1633–1655.

61. Le Tien-Duy B., Oentaryo Richard Jayadi, Lo David. Information retrieval and spectrum based bug localization: better together. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015:579–590; 2015.

62. Hong Shin, Lee Byeongcheol, Kwak Taehoon, et al. Mutation-Based Fault Localization for Real-World Multilingual Programs. In: Cohen Myra B., Grunske Lars, Whalen Michael, eds. *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, :464–475IEEE Computer Society; 2015.

63. Hong Shin, Kwak Taehoon, Lee Byeongcheol, et al. MUSEUM: Debugging real-world multilingual programs using mutation analysis. *Information & Software Technology*. 2017;82:80–95.

64. Moon Seokhyeon, Kim Yunho, Kim Moonzoo, Yoo Shin. Ask the Mutants: Mutating Faulty Programs for Fault Localization. In: :153–162IEEE Computer Society; 2014.

65. Papadakis Mike, Traon Yves Le. Effective fault localization via mutation analysis: a selective mutation approach. In: Symposium on Applied Computing, SAC 2014:1293–1300; 2014.

66. Papadakis Mike, Traon Yves Le. Metallaxis-FL: mutation-based fault localization. *Softw. Test., Verif. Reliab.*. 2015;25(5-7):605–628.

67. Zhang Lingming, Zhang Lu, Khurshid Sarfraz. Injecting mechanical faults to localize developer faults for evolving software. In: :765–784ACM; 2013.

68. Hao Dan, Zhang Lu, Xie Tao, Mei Hong, Sun Jiasu. Interactive Fault Localization Using Test Information. *J. Comput. Sci. Technol.*. 2009;24(5):962–974.

69. Hao Dan, Zhang Lingming, Zhang Lu, Sun Jiasu, Mei Hong. VIDA: Visual interactive debugging. In: :583–586; 2009.

70. Lin Yun, Sun Jun, Xue Yinxing, Liu Yang, Dong Jin Song. Feedback-based debugging. In: Uchitel Sebastián, Orso Alessandro, Robillard Martin P., eds. *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, :393–403; 2017.

71. Ko Andrew Jensen, Myers Brad A.. Debugging reinvented: asking and answering why and why not questions about program behavior. In: Schäfer Wilhelm, Dwyer Matthew B., Gruhn Volker, eds. *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, :301–310; 2008.

72. Li Xiangyu, Zhu Shaowei, dâĂŹAmorim Marcelo, Orso Alessandro. Enlightened Debugging. In: Proceedings of the ACM/IEEE International Conference on Software Engineering:82–92; 2018; New York, NY, USA.

73. Xia Xin, Bao Lingfeng, Lo David, Li Shanping. "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 2016:267–278; 2016.

74. Baudry Benoit, Fleurey Franck, Le Traon Yves. Improving Test Suites for Efficient Fault Localization. In: Proceedings of the 28th International Conference on Software Engineering:82–91; 2006; New York, NY, USA.

75. Campos José, Abreu Rui, Fraser Gordon, d'Amorim Marcelo. Entropy-based Test Generation for Improved Fault Localization. In: Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering:257–267; 2013; Piscataway, NJ, USA.

76. Gonzalez-Sanchez Alberto, Piel Éric, Abreu Rui, Gross Hans-Gerhard, Gemund Arjan J. C.. Prioritizing Tests for Software Fault Diagnosis. *Softw. Pract. Exper.*. 2011;41(10):1105–1129.

77. González-Sanchez Alberto, Abreu Rui, Gross Hans-Gerhard, Gemund Arjan J. C.. Prioritizing tests for fault localization through ambiguity group reduction. In: 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011):83–92; 2011.

78. Dandan Gong, Tiantian Wang, Xiaohong Su, Peijun Ma. A Test-suite Reduction Approach to Improving Fault-localization Effectiveness. *Comput. Lang. Syst. Struct.*. 2013;39(3):95–108.

79. Hao Dan, Xie Tao, Zhang Lu, Wang Xiaoyin, Sun Jiasu, Mei Hong. Test input reduction for result inspection to facilitate fault localization. *Autom. Softw. Eng.*. 2010;17(1):5–31.

80. Xuan Jifeng, Monperrus Martin. Test case purification for improving fault localization. In: Cheung Shing-Chi, Orso Alessandro, Storey Margaret-Anne D., eds. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, :52–63ACM; 2014.

81. Masri Wes, Assi Rawad Abou. Cleansing Test Suites from Coincidental Correctness to Enhance Fault-Localization. In: Third International Conference on Software Testing, Verification and Validation, ICST 2010:165–174; 2010.

82. Perez Alexandre, Abreu Rui, Deursen Arie. A test-suite diagnosability metric for spectrum-based fault localization approaches. In: Proceedings of the 39th International Conference on Software Engineering, ICSE 2017:654–664; 2017.

83. Xie Xiaoyuan, Wong W. Eric, Chen Tsong Yueh, Xu Baowen. Metamorphic slice: An application in spectrum-based fault localization. *Information & Software Technology*. 2013;55(5):866–879.