Division by Zero: Threats and Effects in Spectrum-Based Fault Localization Formulas

Dániel Vince, Attila Szatmári, Ákos Kiss, and Árpad Beszedes
Department of Software Engineering, University of Szeged, Szeged, Hungary
{vinced, szatma, akiss, beszedes}@inf.u-szeged.hu

***Abstract*—Spectrum-Based Fault Localization (SBFL) is based on risk formulas to rank program elements, which work generally well in various situations. However, it cannot be ruled out that zero division might happen during score calculation, which has negative consequences, e.g., essential elements will not be in the top part of the rank list. The literature has given several strategies to tackle the problem, although there is little knowledge on which one to use. In our work, we performed mathematical analysis and an empirical study to find out how this phenomenon affects SBFL. Results show that division by zero happens in many cases, and the strategies can mitigate their consequences with varying success. Thus, we propose a combined method to avoid the threat of division by zero and improve the trustworthiness of SBFL. Our proposals should be taken into consideration whenever a formula is being used or a new one is proposed.**

***Keywords*—Debugging, Division by Zero, Empirical, Fault Localization, Spectrum-based Fault Localization**

I. Introduction

Debugging is one of the most effort intensive activities during development and maintenance, and the brunt of debugging is generally seen to be localizing the fault. Like for many tasks in software maintenance, it is also true for fault localization that the more automated the better.

A widespread idea to localize faults automatically is based on program spectra [1], [2], i.e., on information about program execution during the testing phase. This information usually includes if or how often program elements (statements, branches, function calls, etc.) are executed during each test case and this field of research is called *spectrum-based fault localization* (SBFL). The intuition behind using the program spectra is that those program elements that are exercised by more failing tests than passing ones are more likely to contain the fault. The state-of-the-art techniques [3] use *hit-based* spectra, i.e., binary information about the execution of elements recorded on a suite of passing and failing test cases. Statistics can be derived from the produced binary matrices, i.e., how many passing and failing executions covered or missed the elements, which can then be used for further calculations, such as computing the suspiciousness score for each program element and use it for ranking. A good SBFL technique is expected to give a high rank to the faulty element (ideally, the 1st place), thus guiding the debugging efforts of the software engineer.

However, even the best fault localization techniques are in trouble when division by zero occurs. Several techniques came from the fields of biological research [4]–[6] and have not been prepared for the challenges posed by software engineering, especially automated fault localization. The reason why the division by zero problem exists varies from formula to formula. However, a typical case is when the spectrum of a program element is skewed, i.e., it has been exercised either by failing or by passing tests only. Another extreme case is when a program element has not been exercised at all or it has been exercised by all test cases, which can also result in a division by zero in some formulas. Furthermore, it can also happen that not the spectrum of a program element but the whole test suite is unbalanced, e.g., in fuzz testing, when the discovered bug is triggered by only one (generated) test case and other tests, if they exist, come from a regression test suite that do not signal the failure at all.

A detailed discussion about division by zero occurrences can be found in the following sections of this paper, however, it can already be seen that treating different cases uniformly would result in inefficient fault localization, which may lead to a loss of confidence in the technique. Motivated by these issues, our goal is to answer the following research questions:

Research Questions

RQ1. How can mathematical analysis classify SBFL formulas in terms of division by zero?
RQ2. How often does division by zero happen in practice?
RQ3. Can enhancements be proposed to the formulas in order to improve the effectiveness of SBFL algorithms to localize bugs more successfully?
RQ4. If the score calculation is corrected, to what extent do non-faulty program elements affect the ranking of the faulty program elements?

Thus, in this paper, we utilize mathematical analysis to gather more information about the published SBFL techniques in the last 3 to 5 decades, and whether and how division by zero affects them (Section III). Furthermore, based on the analysis, we propose to categorize the collected formulas into classes and give a reasonable, useful, context-dependent solution to avoid division by zero for each problematic formula class (Section IV). Additionally, we conduct an empirical study on the effects of division by zero on the score calculation in SBFL formulas using the Defects4J and JerryScript datasets (Sections V and VI). Therefore, we guarantee that the existing formulas work even in corner cases, like division by zero.

2693-9177/22/$31.00 ©2022 IEEE
DOI 10.1109/QRS57517.2022.00032

II. Background

A. Spectrum-Based Fault Localization

Given the elements of a program, $|\{e_j\}| = n$, and test cases as inputs, $|\{t_i\}| = m$, a program element hit spectrum is a binary matrix, $\mathbf{S} = (s_{ij}) \in \mathbb{B}^{m \times n}$, where each element of the matrix denotes whether the execution of the program on test input t_i has covered program element e_j. Usually, $s_{ij} = 1$ denotes that the program element has been covered and $s_{ij} = 0$ otherwise. The hit spectrum is usually accompanied by a binary result vector, $\mathbf{R} = (r_i) \in \mathbb{B}^m$, where each element denotes whether the execution of the program on test input t_i has resulted in a failure ($r_i = 1$) or not ($r_i = 0$). A typical representation of these structures is shown below:

$$\mathbf{S} = \begin{array}{c} \\ t_1 \\ t_2 \\ \vdots \\ t_m \end{array} \begin{array}{c} \begin{array}{cccc} e_1 & e_2 & \cdots & e_n \end{array} \\ \begin{pmatrix} 0/1 & 0/1 & \cdots & 0/1 \\ 0/1 & 0/1 & \cdots & 0/1 \\ \vdots & \vdots & \ddots & \vdots \\ 0/1 & 0/1 & \cdots & 0/1 \end{pmatrix} \end{array} \qquad \mathbf{R} = \begin{pmatrix} 0/1 \\ 0/1 \\ \vdots \\ 0/1 \end{pmatrix}$$

Let e denote a program element, usually a function or a statement, then using the $\mathbf{S}$ and $\mathbf{R}$ matrices, four derived values can be calculated per program element e, presented with the following notations:

- $c_{ef}(e)$: number of *failing* test cases executing e,
- $c_{nf}(e)$: number of *failing* test cases not executing e,
- $c_{ep}(e)$: number of *passing* test cases executing e, and
- $c_{np}(e)$: number of *passing* test cases not executing e.

The majority of the published SBFL formulas in the literature, and all that we investigate in this paper, use these $c_{..}(e)$ values to calculate suspiciousness scores in various ways (although notation may vary across papers). Some of them use all of the four values, but there are some that use only a subset. Note that $c_{ef}(e)+c_{nf}(e)$ and $c_{ep}(e)+c_{np}(e)$ are the same for all program elements, giving the number of *failing* and *passing* test cases, c_{fail} and c_{pass}, respectively. Also note that for the sake of simplicity we omit the parametrization with (e) from the notation in the rest of the paper.

B. SBFL Formulas and Division by Zero

Formulas are used to take the four derived values – introduced above – and calculate a suspiciousness score for each program element. When a score has been assigned to all program elements, they are prioritized according to the score in descending order. The highest ranked element is considered the most suspicious, which should be investigated by the software engineer. Over the years, multiple formulas have been proposed to rank program elements in order to predict the faulty statements more accurately. Several of these formulas have been adapted from biological and medical research studies to computer science, e.g., [7]. In their original research domains, all their $c_{..}$ values are strictly positive, so division by zero most probably cannot happen. However, in computer science and more specifically in fault localization, the above assumption does not necessarily hold and division by zero can happen, and unfortunately not all of the formulas h been adapted to this case. Thus it is worth analyzing them from the division by zero point of view in their new environment. We have collected formulas from various sources [3], [6], [8]–[16] and investigated them from a practical perspective: what is their relation to division by zero?

Still, there are some assumptions that can be made during the analysis of the collected formulas in the SBFL use case:

- $\forall c \geqslant 0$: all $c_{..}$ values must be greater than or equal to zero,
- $c_{ef} + c_{nf} = c_{fail} > 0$: the number of *failing* test cases must be at least one, otherwise there would be nothing to localize, and
- $c_{ep} + c_{np} = c_{pass} \geqslant 0$: the number of *passing* test cases can be zero, e.g., this can be the case when a regression test suite is not available, only a (potentially randomly generated) *failing* test case [17].

In terms of the previously introduced matrix notation, column vectors in $\mathbf{S}$ can be zero (i.e., for a given element e_j, $\forall_i : s_{ij} = 0$) if and only if none of the test cases covered an element of the software under test (SUT). Additionally, row vectors in $\mathbf{S}$ can also be zero (i.e., for a given test case t_i, $\forall_j : s_{ij} = 0$), which indicates that a test case has not covered any of the elements in the SUT. This can happen in software systems where only parts of the code base (e.g., the core logic) are compiled with support for coverage information, however, the execution is halted before reaching these parts.

These assumptions have been taken into account during the mathematical analysis. Especially $c_{fail} > 0$ narrowed down the search space, since many formulas use division by $c_{ef} + c_{nf}$, which is strictly greater than zero.

C. Related Research

Numerous automatic fault localization techniques have been published in the last decades, and several surveys and empirical studies [9], [10], [12], [16], [18]–[20] collected them. However, only a few of the published techniques addressed the division by zero problem explicitly, thus it remained an open question in the research domain. Several formulas came from biological research where division by zero cannot happen due to the properties of the processed data, therefore, the authors did not have to deal with the problem. After their adaptation to software maintenance, a few general guidelines have been published about what to do when the problem occurs, but these guidelines typically did not consider the specifics of the formulas.

The first – and most naive – approach might be to assign a zero suspiciousness score to the program element if any of the denominators in the formula evaluates to zero as discussed by Sarhan and Beszédes [21]. Beyond this solution, several proposals have been made for more efficient operation. A popular method is proposed by Jones and Harold [22] that assigns zero to any division in the formula if its denominator evaluates to zero, thus the suspiciousness score can be calculated as if the division did not exist. Similar to that, Yoo [23] used the constant value 1 for this purpose. Naish *et al.* [24] assigned zero to the whole formula if the numerator was also zero and a suitably large value (N) otherwise. This suitably

large value is expected to be larger than any value that can be returned with a non-zero denominator, e.g., the number of tests plus 1 with the Overlap formula or numerator plus 1 with the DStar formula. As an alternative, both Naish *et al.* [24] and Lee [25] have considered adding a small ϵ value to the denominator to avoid division by zero. Lee and Naish [26] considered $\frac{x}{0} = 9999$ and $\frac{0}{0} = 0.5$. Furthermore, Xue and Namin [27], and Landsberg *et al.* [28] have decided to add ϵ to each of the $c_{..}$ coefficients. These approaches can be grouped as follows:

1) assign a predefined value to the score,
2) add a suitably small value (ϵ) to the denominator, or
3) add a suitably small value (ϵ) to all coefficients.

Vince *et al.* briefly analyzed the problem in their study [17] and concluded that the approaches can be formalized by modifying the division operation as

$$\mathrm{div}^{\langle a,b\rangle}(x,y) = \begin{cases} a & \text{if } x = 0 \wedge y = 0 \\ b & \text{if } x \neq 0 \wedge y = 0 \\ x/y & \text{otherwise} \end{cases}$$

or as

$$\mathrm{div}_{\langle +d\rangle}(x,y) = \frac{x}{y+d}$$

or by modifying the values of the coefficients $c_{..}$, denoted as $c + \epsilon$.

Table I summarizes the existing solutions and how they handle division by zero. The first column formulates the solution using the above-defined notation and the second column references the papers that proposed the method.

TABLE I: Existing Solutions for Division by Zero

Formula	**References**
$\mathrm{div}^{\langle 0,0\rangle}$	Jones and Harrold [22], Lee [25]
$\mathrm{div}^{\langle 0,N\rangle}$	Naish *et al.* [24]
$\mathrm{div}^{\langle N,N\rangle}$	Lee [25]
$\mathrm{div}^{\langle 1,1\rangle}$	Yoo [23]
$\mathrm{div}^{\langle 0.5,9999\rangle}$	Naish and Lee [26]
$\mathrm{div}^{\langle 0,1\rangle}$	Troya *et al.* [14]
$\mathrm{div}_{\langle +\epsilon\rangle}$	Naish *et al.* [24], Lee [25]
$c + \epsilon$	Xue and Namin [27], Landsberg *et al.* [28]

D. Tool Implementations

Motivated by the literature, several IDE-integrated or command-line-based fault localization tools have been implemented. They usually provide an infrastructure to automatically instrument the source code of the SUT in order to generate the spectra of the program executions. Furthermore, they often use colors to mark the program elements from green to red based on their suspiciousness scores. Most of the publicly available, open-source implementations are targeting different programming languages or frameworks, however, all of them face the division by zero problem.

Campos *et al.* [29] proposed a tool called "GZoltar"[1] that adopts SBFL to Java. It is available as a command-line tool, as a plug-in for Maven, and as an extension for Eclipse and Visual Studio Code, and it employs several methods to calculate suspiciousness scores. Their first line of defense against division by zero is checking if the spectrum matrices are valid, and if not, zero is assigned as the score. Then, if a formula for a program element evaluates its denominator as zero, the score for that element will be one as Yoo [23] suggested. Ribeiro *et al.* [30] proposed another tool for Java-based programs, called "Jaguar"[2], that assigns zero as the score for those program elements that did not contribute to the failing program execution ($c_{ef} = 0$) and also uses zero as the default value if division by zero happens. Defaulting to zero is popular among the tools, the approach is also used by the "flacoco"[3] tool of Silva *et al.* [31] and by "CharmFL"[4] by Sarhan *et al.* [32].

The published studies try to solve the existing problem in the most generic way, and the publicly available tools use the most naive solutions. Our experimental results show that different formulas work best with different approaches, and there is room for context-dependent optimizations.

III. Mathematical Classification

In order to get a good overview of the field of research, we collected SBFL formulas from several sources [3], [6], [8]–[16], including only those that contain division while excluding those that contain additions, subtractions, or multiplications only, since they do not fall within the scope of this study. Regardless of the notation used in the original publication of the formulas, we converted them to the common notation described in Section II-A to facilitate their analysis. Then, they were analyzed from a pure mathematical perspective with the help of limit calculation. We were interested in how the formulas behave when the divisor tends to zero, without applying any context-dependent knowledge. The collected formulas can be found in Appendix[5].

Eventually, the formulas have been sorted into three categories: division by zero cannot happen (discussed in Section III-A), the limit can be determined exactly – Section III-B, and the limit cannot be determined mathematically – Section III-C. Each section gives at least one example formula that falls into that class.

A. Division by Zero Cannot Happen

Some formulas have been constructed in a way that they contain denominators that cannot evaluate to zero. This can come from the addition of a positive constant to the denominator or from the previously mentioned assumptions, e.g., at least one failing test case is required to execute fault localization, formally $c_{ef} + c_{nf} > 0$. An example of this category is the Jaccard formula [3], which assigns a suspiciousness score

[1]https://github.com/GZoltar/gzoltar, git hash: 310fba7.
[2]https://github.com/saeg/jaguar, git hash: efe8716.
[3]https://github.com/SpoonLabs/flacoco, git hash: 2a74fda.
[4]https://github.com/sed-szeged/CharmFL, git hash: 40dd511.
[5]https://doi.org/10.6084/m9.figshare.21071278

to each program element that is proportional to how many times it has contributed to failing test cases and inversely proportional to the total number of failing test cases and contributions to the passing test cases.

$$Jaccard(e) = \frac{c_{ef}}{c_{ef} + c_{nf} + c_{ep}}$$

The expression in the denominator contains $c_{ef} + c_{nf}$, the number of failing test cases that cannot be zero ($c_{fail} > 0$), thus the denominator never evaluates to zero even if $c_{ep} = 0$ holds. This class contains 25 formulas from the investigated 75, which are shown in Table 1 of the online Appendix[5].

B. Limit Can Be Determined

Formulas in this category have the specific property that their limits can be determined mathematically. An example of this class is DStar [33] (D^*), which divides by zero when $c_{nf} + c_{ep} = 0$.

$$\lim_{(c_{nf}, c_{ep}) \to (0,0)} \frac{c_{ef}^*}{c_{nf} + c_{ep}} = \infty$$

In that specific scenario, the formula tends to ∞. The ∞ as limit might have the meaning that the program element is very suspicious (which is the case with the D^*), however, this depends on the formula and we cannot state that as the ground truth. Another example is CBI_{SQRT} [8], which approaches 0 and is defined as follows:

$$\lim_{(c_{ef}, CBI_{INC}) \to (0,0)} \frac{2}{\frac{1}{CBI_{INC}} + \frac{\sqrt{c_{ef} + c_{nf}}}{\sqrt{c_{ef}}}} = 0$$

It uses another formula, called CBI_{INC}, which, for the purpose of the analysis, has been considered as a coefficient that can take the value of 0. This class includes 21 formulas that are shown in Table 2 of the online Appendix[5].

C. Limit Does Not Exist

A common pattern in the investigated formulas can be described as follows:

$$\frac{x}{x + y}$$

The limit of this fraction does not exist, since the solutions are different depending on the direction in which the fraction is approaching to zero, i.e.,

$$\lim_{(x,0) \to (0,0)} \frac{x}{x + y} = \lim_{(x,0) \to (0,0)} \frac{x}{x + 0} = 1$$

$$\lim_{(0,y) \to (0,0)} \frac{x}{x + y} = \lim_{(0,y) \to (0,0)} \frac{0}{0 + y} = 0$$

This pattern is common among the collected fault localization formulas. An example formula for this class is Tarantula [3], which performs division by zero when $c_{ep} + c_{np} = 0$, i.e., the test suite does not contain *passing* test cases.

$$Tarantula(e) = \frac{\frac{c_{ef}}{c_{ef} + c_{nf}}}{\frac{c_{ef}}{c_{ef} + c_{nf}} + \frac{c_{ep}}{c_{ep} + c_{np}}}$$

Here, the limit of the error-causing part cannot be determined unambiguously, and for this reason, the limit of the whole formula cannot be determined either. There are 29 formulas that belong to this category.

Answer to Research Question #1

Three disjunct categories can be created from the examined formulas. Division by zero cannot happen (25 formulas), the limit exists (21), and the limit cannot be determined (29).

IV. Practical Improvements

In this section, we discuss context-dependent aspects of the problem, i.e., in practice we may assume some restrictions that the mathematical analysis might not consider.

In most systems, a typically constructed test case does not exercise all of the program elements, therefore, it may happen that a program element has not been exercised at all ($\exists e_j : c_{ef}(e_j) + c_{ep}(e_j) = 0$ holds). Based on our analysis, 33 formulas perform division by zero when $c_{ef} + c_{ep} = 0$ (for three, even $c_{ef} = 0$ can trigger division by zero, for one, $c_{ep} = 0$ is enough). Even if division by zero does not happen, it is an important corner case in this research area. We can argue that parts of the SUT that are not executed rarely contribute to the faulty behavior, i.e., typically only the executed code parts contribute to the failure. For these reasons, our ***first proposal*** is to exclude those program elements from the ranking process where $c_{ef} + c_{ep} = 0$ holds. Formulas should not even be applied to program elements that do not satisfy this restriction. Applying this proposal excludes many of the potential division by zero cases, but not all. Table II shows 28 formulas that are constructed in a way that are still affected by zero division. (We have analyzed the source code of several open-source tools – discussed in Section II-C –, however, contrary to our expectations, none of them excluded the non-executed program elements from the ranking.)

TABLE II: Division by Zero Can Happen After the Exclusion of Non-Executed Program Elements

Formulas		
Ample	Gower	Pierce
Ample 2	Gower 3 (YuleQ)	Rogot
Arithmetic Mean	Harmonic Mean	Rogot 2
CBI Log	Kulczynski	Scott
CBI Sqrt	M1	Tarantula
Cohen	Minus	Scott
Collective Strength	Mountford	YuleY
Conviction	Ochiai 2	Zoltar
DStar	Overlap	
Fleiss	Pearson	

Division modes discussed in the literature and summarized in Table I are general guidelines without knowledge about the underlying formula. However, different zero division avoidance methods might work best for different formulas. ıs, our ***second proposal*** is to find the best performing

division mode for each formula on an empirical basis (but especially for those where mathematical analysis does not give an unambiguous answer). The results of such an investigation are shown in Section VI, after the evaluation of our empirical results. As new formulas are published from time to time in order to improve SBFL, we suggest for future authors to include a detailed zero division avoidance guide or to use a suitable division mode from Table I, preferably justified by mathematical analysis or by empirical results, whenever an affected formula is proposed.

Discussing further the common behavior of faults, program elements that are exercised in every test execution have a smaller chance of containing the bug. Several well-performing formulas use this intuition, i.e., reducing the score by some proportion, however, not all of them. Some of the investigated formulas contain this exact expression, e.g., Conviction, thus evaluating it to zero may result in a division by zero. However, exclusion of those program elements where $c_{nf}+c_{np}=0$ holds is not safe, and cannot be performed without a potential to lose information.

V. Empirical Evaluation

To evaluate the effect of different division modes and the proposed improvement ideas, we evaluated 434 faulty program versions from the Defects4J [34] dataset[6] and 11 tests from the JerryScript Reduction Test Suite (JRTS)[7]. Defects4J (v1.5) contains 438 reproducible bugs from 6 real-world open source programs (approximately 27.5K methods and 330K LOC), and each is accompanied by a test suite that can expose that bug. JRTS contains 13 fuzzer-generated test inputs (JavaScript sources) that trigger failures in various versions of the JerryScript[8] engine (approximately 1.5K methods and 122K LOC). We used both datasets to empirically study how the above-discussed division modes affect the score calculation of the 75 collected formulas. Furthermore, we investigated whether our proposals help the algorithms be more successful at localizing bugs. Additionally, we combined the categories from our mathematical classification with the existing division modes.

Parnin *et al.* [35] showed that statement-level fault localization might be too fine-grained and miss useful context information. On the other hand, Wong *et al.* [3] showed class level fault localization is too coarse-grained and does not help the developer understand and fix the fault within a class. Hence, we used *method-level* granularity in this study. In order to compare the effectiveness of the different division modes, we compiled the test programs with coverage information, then executed the tests to get the the program spectra (the $\mathbf{S}$ and $\mathbf{R}$ matrices from Section II). Then, we calculated the $c_{..}(e)$ coefficients for all program elements, which are the inputs of the formulas. With this information, the scores and the ranks could be calculated with all the formulas and all the division modes within them.

[6]https://github.com/rjust/defects4j/tree/v1.5.0

[7]https://github.com/vincedani/jrts/commit/7c6f8f

[8]https://github.com/jerryscript-project/jerryscript

Having the suspiciousness ranking list made, we can conclude which modes in Table I are better than the rest. A division mode is more successful in helping the algorithm find the fault if the faulty element is nearer to the top in the list of suspicious elements. The *rank position* is an efficient way to compare the effectiveness of algorithms, because it gives clear information about the effort the developer has to put into investigating the elements in the list. Several studies [36], [37] investigated the trustworthiness of fault localization and concluded that the buggy element must appear in the top-5 positions in the rank list to be investigated by the engineer repairing the software. Agreeing with these results, we consider a formula effective when it puts the faulty element within the top-5 positions in the rank list. Therefore, we consider it an *enabling improvement* whenever faulty elements go into the top-5 list after the selected division mode changes their score. However, to get more detailed results, we also investigate the top-1 and top-3 results of the ranking.

In order to see whether division modes make a difference in score calculation and thus change the ranking list, we need to know how often division by zero happens for each investigated formula. The answer to this question determines whether our study has relevance at all. The more problems formulas have the more chance we have for enhancing the ranking by using different division modes.

Furthermore, the division by zero problem might affect the majority of elements during score calculation, not just faulty ones. Therefore, the question arises: does defining a suspiciousness score for all affected elements indeed improves the rank of the faulty element (considering that non-faulty elements may get a new score assigned, too)? Not to mention the problem that in real fault localization scenarios, we do not have prior knowledge about which program elements are faulty, thus we have to deal with both cases in this study.

VI. Results and Discussion

A. Occurrences of Division by Zero

First, we calculated the scores for all 75 collected formulas to cross-validate whether the results of mathematical analysis align with practice and see how often division by zero causes problems. Table III shows the frequency of zero division during score calculation on the two benchmark suites (i.e., we investigated how many times division by zero happened during calculating the scores of the program elements in all faulty program versions). Formulas which belong to the "*Division by Zero Cannot Happen*" class are not included, so this left us with 39 formulas that turned out to be affected. These are grouped by their categories based on the mathematical analysis. We made a sanity check that the not included formulas indeed did not cause the division by zero problem in practice. The "All" columns contain how many times division by zero occurred with any program element and columns "Faulty" contain the same results for the faulty elements only.

We identified five formulas, highlighted with red, i.e., CBI Log, CBI Sqrt, Conviction, Overlap and Zoltar, that are heavily affected. But other formulas are also affected as the number livisions by zero typically falls between 3 and 30 for faulty

TABLE III: Frequency of Division by Zero in the Used Benchmark Programs

Metric	Defects4J All	Defects4J Faulty	JerryScript All	JerryScript Faulty
Limit exists				
CBI Inc (CBI)	625,089	5	2,130	0
CBI Log	725,018	408	6,030	10
Conviction	752,560	525	6,030	10
CorRatio (Ochiai 3)	625,089	5	2,130	0
Dennis	625,089	5	2,130	0
DStar	109	25	50	0
Fager	625,089	5	2,130	0
Fossum	625,089	5	2,130	0
Gower	625,180	5	3,539	3
Gower 3 (YuleQ)	625,180	5	3,539	3
GP 13	625,089	5	2,130	0
Hyperbolic	625,089	5	2,130	0
Kulczynski	109	25	50	0
M1	109	25	50	0
Ochiai	625,089	5	2,130	0
Pearson	625,180	5	3,539	3
YuleV	625,089	5	2,130	0
YuleY	625,180	5	3,539	3
Zoltar	2,616,037	69	11,816	2
Limit does not exist				
AssocDice	625,089	5	2,130	0
Barinel (Coef, SBI)	625,089	5	2,130	0
CBI Inc (CBI)	625,089	5	2,130	0
Certainty	625,089	5	2,130	0
Collective Strength	26	0	1,410	3
Confidence	625,089	5	2,130	0
Correlation	625,089	5	2,130	0
Forbes	625,089	5	2,130	0
Gower 3 (YuleQ)	625,180	5	3,539	3
Harmonic Mean	625,180	5	3,539	3
Hyperbolic	625,089	5	2,130	0
Interest	625,089	5	2,130	0
Klosgen	625,089	5	2,130	0
Kulczynski 2	625,089	5	2,130	0
McCon	625,089	5	2,130	0
Minus	625,180	5	3,539	3
Mountford	625,198	30	2,180	0
Ochiai2	625,180	5	3,539	3
Overlap	2,743,508	589	15,716	12
Pierce	200	25	1,489	3
Rogot2	625,180	5	3,539	3
Tarantula	625,089	5	2,130	0

Color code: *white*: low frequency, *yellow*: medium frequency, and *red*: high frequency.

elements, and between 2K and 625K for non-faulty ones (these cases are highlighted with yellow). When comparing how faulty and non-faulty program elements are affected, division by zero caused a lot fewer problems for the faulty elements than for the non-faulty ones. This is not surprising, since the distribution of the two categories, i.e., faulty or non-faulty, is uneven. However, the problem during score calculation non-faulty elements affect the ranking of faulty ones.

Answer to Research Question #2

- Formulas that got marked as safe during mathematical analysis (25 items) did not perform any division by zero during the experiments.
- For 39 of the 50 remaining formulas, division by zero is an existing problem during score calculation in the used benchmark programs, both with faulty and non-faulty program elements, proving that our study has relevance.

B. Impact of the Proposed Enhancements

We evaluated the division modes presented in Table I on each buggy version of programs from both datasets to determine which one performs best with the collected formulas. Section II-C contains the analysis of several open-source fault localization tools that handle division by zero in the simplest possible way, assigning zero to the whole formula when the problem occurs. For this reason, we consider this approach as our baseline and compare the different division modes to it. To compare the division modes, we needed to calculate the average of average ranks of the 39 formulas (for both datasets) that are affected by the problem using each division mode. We consider a division mode *better* than the others if the ranks of the faulty elements are closer to the top of the rank list with its usage.

Table IV contains detailed information – with data from both test programs – about how many times each formula used zero as the denominator during processing the program spectra. Program elements that were not executed by any of the tests were excluded from the ranking, thus many division by zero occurrences disappeared from the score computation. However, the "Frequency of Division by Zero" column shows that there is still a large number of zero divisions, which makes the division mode selection necessary. Some items are missing from the table compared to Table II, which means that not all formulas that were analyzed and marked as *unsafe* caused division by zero in the used programs. The column also contains relative differences compared to Table III and shows that the majority of the zero divisions are eliminated by this proposed enhancement in 11 of 19 cases. Those formulas where the issue could not be eliminated to any extent are denoted with *(—)* as the relative change is zero.

For the remaining cases, it can be determined which mode results in the most advantageous rank list in terms of top-5 rank. For each formula, scores and ranks are computed with different division modes and grouped by division mode. The column "Best Performing Division Mode" contains which mode performed the best for each metric. The two outstanding modes are the $\mathrm{div}_{\langle+\epsilon\rangle}$ and the $c + 0.1$, giving the best results in 8 cases each. Formulas in Table IV have unique internal structures, and Section III discussed that some of them are approaching an exact limit when the denominator is approaching zero. However, the empirical results show that modifying the division or adding a small ϵ to the coefficients

TABLE IV: Effect of Proposed Enhancements in the Used Benchmark Programs

Name	Frequency of Division by Zero		Best Performing Division Mode	Enabling Improvement		Formula
CBI Log	103,829	*(-85.80%)*	$\text{div}_{\langle+\epsilon\rangle}$	79	*(+58.52%)*	$\frac{2}{\frac{1}{CBI_{Inc}(e)}+\frac{\log(c_{ef}+c_{nf})}{\log(c_{ef})}}$
CBI Sqrt	2,000,134	*(-23.85%)*	$\text{div}_{\langle+\epsilon\rangle}$	0	*(—)*	$\frac{2}{\frac{1}{CBI_{Inc}(e)}+\frac{\sqrt{c_{ef}+c_{nf}}}{\sqrt{c_{ef}}}}$
Collective Strength	1,436	*(—)*	$c+0.1$	0	*(—)*	$1-\frac{c_{ef}+c_{np}}{(c_{ef}+c_{ep})\cdot(c_{ef}+c_{nf})+(c_{nf}+c_{np})\cdot(c_{ep}+c_{np})}\cdot\frac{1-(c_{ef}+c_{ep})\cdot(c_{ef}+c_{nf})-(c_{nf}+c_{np})\cdot(c_{ep}+c_{np})}{1-c_{ef}-c_{np}}$
Conviction	131,371	*(-82.68%)*	$c+0.5$	183	*(+631.03%)*	$\max\left(\frac{(c_{ef}+c_{ep})\cdot(c_{ep}+c_{np})}{c_{ep}},\frac{(c_{ef}+c_{nf})\cdot(c_{nf}+c_{np})}{c_{nf}}\right)$
DStar	159	*(—)*	$\text{div}_{\langle+\epsilon\rangle}$	18	*(+9.00%)*	$\frac{c_{ef}^{*}}{c_{nf}+c_{ep}}$
Gower	1,500	*(-99.76%)*	$c+0.1$	0	*(—)*	$\frac{c_{ef}+c_{np}}{\sqrt{(c_{ef}+c_{nf})\cdot(c_{ef}+c_{ep})\cdot(c_{nf}+c_{np})\cdot(c_{ep}+c_{np})}}$
Gower 3	1,500	*(-99.76%)*	$c+0.1$	133	*(+164.20%)*	$\frac{c_{ef}\cdot c_{np}-c_{nf}\cdot c_{ep}}{c_{ef}\cdot c_{np}+c_{nf}\cdot c_{ep}}$
Harmonic Mean	1,500	*(-99.76%)*	$c+0.1$	0	*(—)*	$\frac{(c_{ef}\cdot c_{np}-c_{nf}\cdot c_{ep})\cdot\left((c_{ef}+c_{ep})\cdot(c_{np}+c_{nf})+(c_{ef}+c_{nf})\cdot(c_{ep}+c_{np})\right)}{(c_{ef}+c_{ep})\cdot(c_{np}+c_{nf})\cdot(c_{ef}+c_{nf})\cdot(c_{ep}+c_{np})}$
Kulczynski	159	*(—)*	$\text{div}_{\langle+\epsilon\rangle}$	18	*(+9.23%)*	$\frac{c_{ef}}{c_{nf}+c_{ep}}$
M1	159	*(—)*	$c+0.1$	21	*(+43.75%)*	$\frac{c_{ef}+c_{np}}{c_{nf}+c_{ep}}$
Minus	1,409	*(-99.78%)*	$c+0.1$	0	*(—)*	$\frac{\frac{c_{ef}}{c_{ef}+c_{nf}}}{\frac{c_{ef}}{c_{ef}+c_{nf}}+\frac{c_{ep}}{c_{ep}+c_{np}}}-\frac{1-\frac{c_{ef}}{c_{ef}+c_{nf}}}{1-\frac{c_{ef}}{c_{ef}+c_{nf}}+1-\frac{c_{ep}}{c_{ep}+c_{np}}}$
Mountford	159	*(-99.97%)*	$\text{div}^{\langle 0,1\rangle}$	18	*(+9.09%)*	$\frac{c_{ef}}{0.5\cdot\left((c_{ef}\cdot c_{ep})+(c_{ef}\cdot c_{nf})\right)+(c_{ep}\cdot c_{nf})}$
Ochiai 2	1,500	*(-99.76%)*	$\text{div}_{\langle+\epsilon\rangle}$	0	*(—)*	$\frac{c_{ef}\cdot c_{np}}{\sqrt{(c_{ef}+c_{ep})\cdot(c_{np}+c_{nf})\cdot(c_{ef}+c_{nf})\cdot(c_{ep}+c_{np})}}$
Overlap	2,132,005	*(-22.73%)*	$\text{div}_{\langle+\epsilon\rangle}$	63	*(+370.59%)*	$\frac{c_{ef}}{\min(c_{ef},c_{nf},c_{ep})}$
Pearson	1,500	*(-99.76%)*	$\text{div}_{\langle+\epsilon\rangle}$	0	*(—)*	$\frac{(c_{ef}+c_{nf}+c_{ep}+c_{np})\left((c_{ef}\cdot c_{np})-(c_{ep}\cdot c_{nf})\right)^2}{(c_{ef}+c_{ep})\cdot(c_{nf}+c_{np})\cdot(c_{ep}+c_{np})\cdot(c_{ef}+c_{nf})}$
Pierce	1,689	*(—)*	$\text{div}^{\langle 1,1\rangle}$	24	*(+800.00%)*	$\frac{(c_{ef}\cdot c_{nf})+(c_{nf}\cdot c_{ep})}{(c_{ef}\cdot c_{nf})+(2\cdot c_{nf}\cdot c_{np})+(c_{ep}\cdot c_{np})}$
Rogot 2	1,500	*(-99.76%)*	$c+0.1$	0	*(—)*	$\frac{1}{4}\cdot\left(\frac{c_{ef}}{c_{ef}+c_{ep}}+\frac{c_{ef}}{c_{ef}+c_{nf}}+\frac{c_{np}}{c_{np}+c_{ep}}+\frac{c_{np}}{c_{np}+c_{nf}}\right)$
YuleY	1,500	*(-99.76%)*	$c+0.1$	133	*(+164.20%)*	$\frac{\sqrt{c_{ef}\cdot c_{np}}-\sqrt{c_{ep}\cdot c_{nf}}}{\sqrt{c_{ef}\cdot c_{np}}+\sqrt{c_{ep}\cdot c_{nf}}}$
Zoltar	2,000,634	*(-23.87%)*	$\text{div}_{\langle+\epsilon\rangle}$	0	*(—)*	$\frac{c_{ef}}{c_{ef}+c_{nf}+c_{ep}+\frac{10000\cdot c_{nf}\cdot c_{ep}}{c_{ef}}}$

leads to better approximation of the scores (than assigning the limit to the score), and hence, to better rankings.

To better understand what happens during the score calculation and the ranking processes, two formulas are investigated further from Table IV. The *Overlap* formula performed the most division by zero operations among the investigated ones, therefore, we paid special attention to it. Figure 1a shows how the ranking is changed as a result of changing the division mode. The green color indicates the number of cases when the formula ranked the faulty program element to the 1st place, then the yellow color indicates that it was ranked in the top-3 places and the cyan means in the top-5. For this formula, the *naive* and the three constant modes i.e., $\text{div}^{\langle 0,0\rangle}$, $\text{div}^{\langle 1,1\rangle}$ and $\text{div}^{\langle N,N\rangle}$, are not effective, but the other division modes boosted the effectiveness to a useful level. The term N was introduced in Section II as a theoretically suitably large value. In our experiments, we used the *infinity* value from the mathematical module of Python 3.

The other formula we investigate is *Conviction*, which has fewer zero values in the denominator in an order of magnitude, but still in hundreds of thousand cases. Division modes behave differently compared to the *Overlap* as shown in Figure although using a division mode is still more effective than assigning zero as the score of the program element (*naive* column).

Table IV shows which different division modes work best for specific formulas. However, in a broader context, the question arises if there a "best division mode" which outperforms the others in terms of average ranking? To be able to analyze the experimental data from this perspective, we averaged the ranks of each input and formula, i.e., the result is the general effect of the division mode. With the exception of $c+0.5$, all modes outperform the *naive* approach, however, the averaged ranks are close to one another, thus there is no "best division mode" that works best in all circumstances. We also aggregated the top-N results of each input with each division mode, however, the same patterns can be observed and a conclusion can be drawn: in general, there is no better or worse division mode and there is no "one fits all solution", i.e., different modes work best with specific formulas.

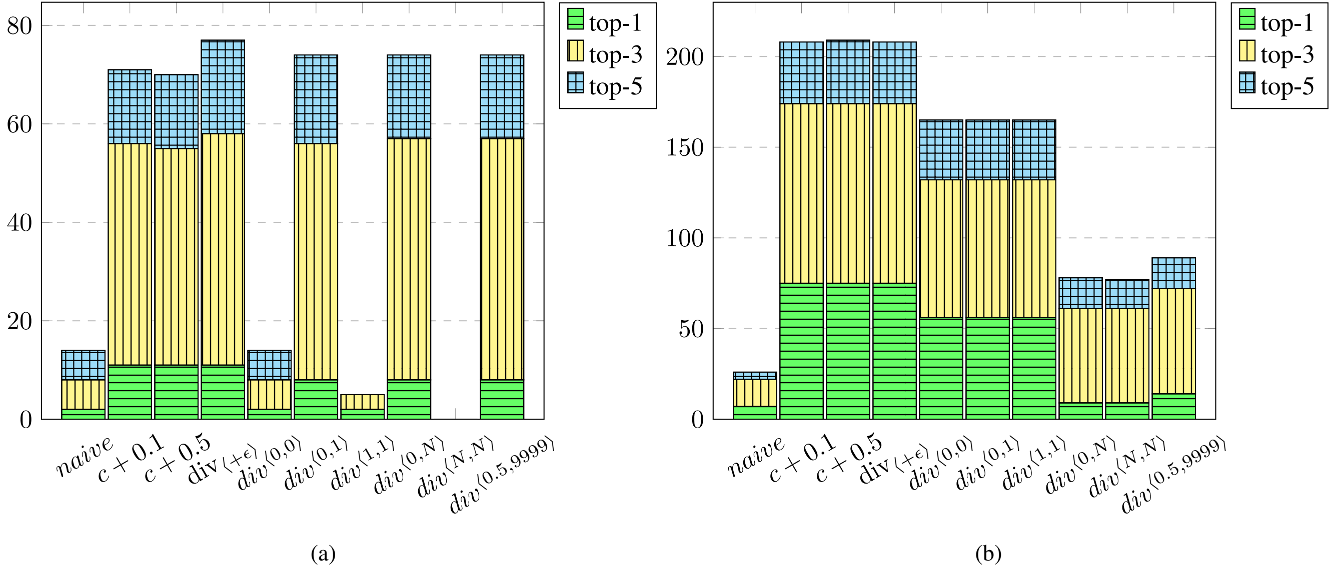

Figure 1: Effects of Different Division Modes on top-5 ranking (the higher the better): (a) Overlap, (b) Conviction.

Answer to Research Question #3

- Excluding those program elements from the score calculation process that are not executed during the testing phase ($c_{ef} + c_{ep} = 0$) reduced the division by zero occurrences (59.84% on average, 99.97% in best case) in our benchmark programs, although the problem still exists to a lesser extent.
- Using any of the already existing division modes instead of assigning zero as a score (marked as *naive* in this study) is beneficial, however, division modes behave differently with various formulas.
- In our benchmark programs, using the best fit division mode for each formula resulted in 27.21% more top-5 ranks on average.

C. Impact of Division by Zero in Non-Faulty Program Elements

In previous sections, we analyzed the impact of the proposed solutions on the faulty elements in terms of changed scores and rank positions. In this section, we focus on the unusual cases, i.e., when this direct relationship does not hold between the score and rank changes. We investigate how the solutions to division by zero problem affect the relationship of *both* the faulty and non-faulty elements together. Our hypothesis is that when different division modes do not affect the score of the faulty element itself, its rank may still change due to changes in the scores of non-faulty elements. Eventually, the rank list depends on the score of each and every program element, not just the faulty ones.

To thoroughly investigate the problem, we need to differentiate between the following four categories:

a) the rank of the faulty element is *worse* (than the baseline) while its score is *better*,

b) the rank of the faulty element is *worse*, although the score is *unchanged*,

c) the rank of the faulty element is *better*, although the score is *unchanged*,

d) the rank of the faulty element is *better* while the score got *worse*,

Category *a)* occurs when the score of the faulty element gets better, however, scores in its environment improve even more, i.e., non-faulty elements will be placed before the faulty using one of the division modes. Categories *b)* and *c)* happen when there is no zero division during the faulty score calculation, however, non-faulty elements will be moved to either before or after the faulty element in the rank list. Finally, the last category *d)* is the polar opposite of *a)*, i.e., even though using a division mode will decrease the scores, the faulty element is still going to be placed in a better position. This happens when the non-faulty scores decrease more than the faulty score.

We counted the four categories using 75 formulas in Defects4J (438 bugs) and JerryScript (13 bugs). Table V shows the percentage of each category in Defects4J and JerryScript. Table V shows that both Defects4J and JerryScript have a relatively high percentage in the *a)* category using $c + 0.1$ and $c + 0.5$ division modes and *b)* category using $div^{\langle N,N \rangle}$. The former happens due to non-faulty elements having better scores to begin with, therefore, the additional ϵ increases the scores even more. In other words, the context of the faulty elements (non-faulty) will pull the faulty element down in the rank list when we use this division mode.

In the latter case, this division mode results in non-faulty elements having better ranks than faulty ones in a relatively high percentage of cases. Moreover, this is one of the reasons why $div^{\langle N,N \rangle}$ strategy is not the best performing for several formulas, as shown in Table IV.

We investigated the division modes using the Overlap formula further, based on the 4 categories. It is not surprising that category *b)* dominates the other categories, while category *a)* has fewer occurrences. This stems from the way the metric is constructed. Overlap's numerator is c_{ef}, however, the

TABLE V: Percentage of categories in Defects4J (*D4J*; 434 bugs) and JerryScript (*JS*; 13 bugs)

Division Mode	**a**		**b**		**c**		**d**	
Dataset	D4J	JS	D4J	JS	D4J	JS	D4J	JS
$c+0.1$	19.2%	19.3%	0.1%	0.0%	0.0%	0.0%	0.4%	0.0%
$c+0.5$	23.6%	21.9%	0.1%	0.0%	0.0%	0.0%	0.5%	0.3%
$\text{div}_{\langle+\epsilon\rangle}$	0.6%	1.8%	3.4%	6.2%	0.1%	0.0%	0.0%	0.0%
$div^{\langle 0,0\rangle}$	0.6%	1.2%	0.7%	1.0%	0.0%	0.0%	0.0%	0.0%
$div^{\langle 0,1\rangle}$	0.7%	1.2%	3.4%	6.3%	0.1%	0.0%	0.0%	0.0%
$div^{\langle 1,1\rangle}$	1.0%	1.2%	18.2%	16.3%	0.0%	0.0%	0.0%	0.0%
$div^{\langle 0,N\rangle}$	0.5%	1.0%	5.8%	9.0%	0.1%	0.0%	0.0%	0.0%
$div^{\langle N,N\rangle}$	0.0%	0.0%	34.1%	28.3%	1.0%	0.9%	0.1%	0.0%
$div^{\langle 0.5,9999\rangle}$	0.8%	1.6%	14.0%	15.8%	0.0%	0.0%	0.0%	0.0%

Categories **a** and **b**: faulty elements get **worse** ranks than non-faulty ones; Categories **c** and **d**: faulty elements get **better** ranks than non-faulty ones.

denominator is a minimum of c_{ef}, c_{nf}, and c_{ep}. Most non-faulty elements have 0 failed test executions, therefore, will produce ties. Otherwise, either c_{nf} or c_{ep} are zero; in that case non-faulty elements may get higher scores, therefore they will be placed before the faulty element in the rank list.

Categories *c)* and *d)* i.e., the faulty element's ranks improve, were practically non-existent in our measurement results, and this means that overall the phenomenon discussed in this section negatively affects the risk formulas' efficiency. However, as seen in previous sections, the overall efficiency of the formulas is affected positively, so this does not invalidate our previous results.

Answer to Research Question #4

The discussed division modes impact all program elements. In the used benchmark programs, using any of the solutions can make the fault localization less effective by putting the non-faulty elements before the faulty ones on the rank list, however, it happens in a relatively few cases only. Division modes $c+0.1$, $c+0.5$, and $div^{\langle N,N\rangle}$ are most affected by this phenomenon.

VII. Conclusions

In this paper, we focus on the problem of division by zero in SBFL formulas. We investigated 75 formulas to find out whether it is an existing problem for most of them. Then, we investigated the research area, how the existing studies deal with zero division. Finally, we analyzed several open-source fault localization tools for different programming languages to check how real applications deal with the problem.

We prototyped the formulas and the existing division modes, then evaluated them on two publicly available benchmarks that have been used in fault localization-related studies. The results of our experiments show that using any division mode instead of assigning zero to the score is beneficial, however, division modes behave differently with various formulas.

We found out that excluding those program elements from the score calculation process that are not executed during the testing phase could eliminate the zero division occurrences in the used benchmark programs by 59.84% on average, although the problem still exists to a lesser extent. For the remaining cases, we proposed to choose empirically the best performing division mode for each affected formula. With this method, SBFL formulas ranked 27% more faulty elements in the top-5 positions of the rank list. In summary, we can formalize the following Proposals:

Proposals of the Study:

Proposal 1. $c_{ef}+c_{ep}>0$: the program element must be executed at least once to get a suspiciousness score,

Proposal 2. If someone uses or plans to use an existing SBFL formula that is affected by the zero division problem, we recommend using the assigned division mode from Table IV to improve the effectiveness of their fault localization.

Proposal 3. If someone designs a new formula, we recommend they include a detailed zero division avoidance guide which will help the developer in utilizing the formula, or use one of the best-performing division modes ($c+0.1$, $\text{div}_{\langle+\epsilon\rangle}$) as a default.

We plan to conduct further experiments to ensure that the results generalize to granularities that are finer or more coarse to those investigated in this paper. We wish to investigate whether different ϵ values can help predicting the faulty statements more precisely in the $\text{div}_{\langle+\epsilon\rangle}$ division mode.

VIII. Acknowledgments

This research was supported by project TKP2021-NVA-09. Project no. TKP2021-NVA-09 has been implemented with the support provided by the Ministry of Innovation and Technology of Hungary from the National Research, Development and ovation Fund, financed under the TKP2021-NVA funding

scheme. Dániel Vince was supported by the ÚNKP-22-3-SZTE-469 New National Excellence Program of the Ministry for Culture and Innovation from the source of the National Research, Development and Innovation Fund.

REFERENCES

[1] T. Reps, T. Ball, M. Das, and J. Larus, "The use of program profiling for software maintenance with applications to the year 2000 problem," *ACM SIGSOFT Software Engineering Notes*, vol. 22, no. 6, pp. 432–449, Nov. 1997.

[2] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi, "An empirical investigation of the relationship between spectra differences and regression faults," *Software Testing, Verification and Reliability*, vol. 10, no. 3, pp. 171–194, 2000.

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

[4] S. Kulczyński, "Die Pflanzenassoziationen der Pieninen," *Bulletin International de l'Académie Polonaise des Sciences et des Lettres, Classe des Sciences Mathématiques et Naturelles, Série B (Sciences Naturelles)*, vol. II, pp. 57–203, 1927.

[5] A. Ochiai, "Zoogeographical studies on the soleoid fishes found in Japan and its neighbouring regions–II," *Bulletin of the Japanese Society of Scientific Fisheries*, vol. 22, no. 9, pp. 526–530, 1957.

[6] R. R. Sokal and P. H. A. Sneath, *Principles of Numerical Taxonomy*. W. H. Freeman and Company, 1963.

[7] A. H. Cheetham and J. E. Hazel, "Binary (presence-absence) similarity coefficients," *Journal of Paleontology*, vol. 43, no. 5, pp. 1130–1136, Sep. 1969.

[8] H. J. Lee. (2013) Survey of software bug localization using dynamic analysis. https://isoft.acm.org/isec2013/docs/tutorial-1.pdf. [Online; accessed 20-July-2021].

[9] S. Heiden, L. Grunske, T. Kehrer, F. Keller, A. van Hoorn, A. Filieri, and D. Lo, "An evaluation of pure spectrum-based fault localization techniques for large-scale software systems," *Software: Practice and Experience*, vol. 49, May 2019.

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

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

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

[13] W. E. Wong, V. Debroy, R. Gao, and Y. Li, "The DStar method for effective software fault localization," *IEEE Transactions on Reliability*, vol. 63, no. 1, pp. 290–308, 2014.

[14] J. Troya, S. Segura, J. A. Parejo, and A. Ruiz-Cortés, "Spectrum-based fault localization in model transformations," *ACM Transactions on Software Engineering and Methodology*, vol. 27, no. 3, pp. 13:1–13:50, Sep. 2018.

[15] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "An evaluation of similarity coefficients for software fault localization," in *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing (PRDC)*. IEEE, Dec. 2006, pp. 39–46.

[16] L. Zhang, L. Yan, Z. Zhang, J. Zhang, W. Chan, and Z. Zheng, "A theoretical analysis on cloning the failed test cases to improve spectrum-based fault localization," *Journal of Systems and Software*, vol. 129, pp. 35–57, 2017.

[17] D. Vince, R. Hodován, and Á. Kiss, "Reduction-assisted fault localization: Don't throw away the by-products!" in *16th International Conference on Software Technologies (ICSOFT 2021)*. SciTePress, 2021, pp. 196–206.

[18] A. Perez, R. Abreu, and A. van Deursen, "A test-suite diagnosability metric for spectrum-based fault localization approaches," in *Proceedings of the 39th International Conference on Software Engineering (ICSE)*. IEEE Press, 2017, pp. 654–664.

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

[20] F. Feyzi and S. Parsa, "Inforence: effective fault localization based on information-theoretic analysis and statistical causal inference," *Frontiers of Computer Science*, vol. 13, no. 4, pp. 735–759, Aug. 2019.

[21] Q. I. Sarhan and A. Beszédes, "A survey of challenges in spectrum-based software fault localization," *IEEE Access*, vol. 10, pp. 10 618–10 639, 2022.

[22] J. A. Jones and M. J. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. ACM, Nov. 2005, pp. 273–282.

[23] S. Yoo, "Evolving human competitive spectra-based fault localisation techniques," in *Search Based Software Engineering – 4th International Symposium, SSBSE 2012, Riva del Garda, Italy, September 28-30, 2012. Proceedings*, ser. Lecture Notes in Computer Science (LNCS), vol. 7515. Springer, 2012, pp. 244–258.

[24] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Transactions on Software Engineering and Methodology*, vol. 20, no. 3, pp. 11:1–11:32, Aug. 2011.

[25] H. J. Lee, "Software debugging using program spectra," Ph.D. dissertation, Department of Computer Science and Software Engineering, The University of Melbourne, 2011.

[26] L. Naish and H. J. Lee, "Duals in spectral fault localization," in *Proceedings of the 2013 22nd Australian Software Engineering Conference (ASWEC)*. IEEE, Jun. 2013, pp. 51–59.

[27] X. Xue and A. S. Namin, "How significant is the effect of fault interactions on coverage-based fault localizations?" in *Proceedings of the 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. IEEE, Oct. 2013, pp. 113–122.

[28] D. Landsberg, H. Chockler, D. Kroening, and M. Lewis, "Evaluation of measures for statistical fault localisation and an optimising scheme," in *Fundamental Approaches to Software Engineering – 18th International Conference, FASE 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings*, ser. Lecture Notes in Computer Science (LNCS), vol. 9033. Springer, 2015, pp. 115–129.

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

[30] H. L. Ribeiro, R. P. A. de Araujo, M. L. Chaim, H. A. de Souza, and F. Kon, "Jaguar: A spectrum-based fault localization tool for real-world software," in *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, 2018, pp. 404–409.

[31] A. Silva, M. Martinez, B. Danglot, D. Ginelli, and M. Monperrus, "Flacoco: Fault localization for java based on industry-grade coverage," *arxiv:2111.12513*, 2021.

[32] Q. Idrees Sarhan, A. Szatmári, R. Tóth, and A. Beszédes, "Charmfl: A fault localization tool for python," in *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2021, pp. 114–119.

[33] W. E. Wong, V. Debroy, Y. Li, and R. Gao, "Software fault localization using DStar (D*)," in *Proceedings of the Sixth IEEE International Conference on Software Security and Reliability (SERE)*. IEEE, Jun. 2012, pp. 21–30.

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

[35] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA '11)*. ACM, 2011, pp. 199–209.

[36] X. Xia, L. Bao, D. Lo, and S. Li, ""Automated debugging considered harmful" considered harmful: a user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems," in *Proceedings of the 2016 IEEE International Conference on Software Maintenance and Evolution (ICSME 2016)*. IEEE, Oct. 2016, pp. 267–277.

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