Reduction-assisted Fault Localization: Don't Throw Away the By-products!

Dániel Vince[a], Renáta Hodován[b] and Ákos Kiss[c]
Department of Software Engineering, University of Szeged, Dugonics tér 13, 6720 Szeged, Hungary

Keywords: Spectrum-based Fault Localization, Test Case Reduction, Fuzz Testing.

Abstract: Spectrum-based fault localization (SBFL) is a popular idea for automated software debugging. SBFL techniques use information about the execution of program elements, recorded on a suite of test cases, and derive statistics from them, which are then used to determine the suspiciousness of program elements, thus guiding the debugging efforts. However, even the best techniques can face problems when the statistics are unbalanced. If only one test case causes a program failure and all other inputs execute correctly, as is typical for fuzz testing, then it may be hard to differentiate between the program elements suspiciousness-wise. In this paper, we propose to utilize test case reduction, a technique to minimize unnecessarily large test cases often generated with fuzzing, to assist SBFL in such scenarios. As the intermediate results, or by-products, of the reduction are additional test cases to the program, we use these by-products when applying SBFL. We have evaluated this idea, and our results show that it can help SBFL precision by up to 49% on a real-world use-case.

1 INTRODUCTION

When a software failure is detected, debugging starts. But to be able to get rid of a bug, it has to be located first. Like for many tasks in the domain of software maintenance, it is also true for *fault localization* that the more automated it is the better.

A popular idea to automatically localize faults is based on program spectrum (Reps et al., 1997; Harrold et al., 2000), on information about the execution of a program from certain perspective (e.g., whether – or how many times – statements, branches, or function call chains are executed), called *spectrum-based fault localization* (SBFL). The state-of-the-art SBFL techniques (Wong et al., 2016) use *hit-based* spectra of program elements – binary information about the execution of statements, blocks, or functions – recorded on a suite of passing and failing test cases, and derive statistics from them (i.e., how many passing or failing executions of the program did or did not cover each of the elements). From these statistics, a so-called suspiciousness score is computed, which is then used to rank the program elements. 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 SBFL techniques are in trouble when the spectra and the statistics are skewed. If only one test case causes a program failure and a lot of other inputs execute correctly, then it may not be easy to differentiate between the program elements suspiciousness-wise. This situation is typical with *fuzzing* or random testing (Takanen et al., 2018), when a newly generated test causes a failure, and what is more, that is the only known failure-inducing input, while all of the existing test suite of the target program passes correctly.

In this paper, we propose to utilize *test case reduction* (Hildebrandt and Zeller, 2000) to assist the localization of faults found with fuzzing. The randomly generated test cases are much larger than necessary by nature, and when one of them triggers a failure, it should preferably be trimmed down to a minimal form. Fortunately, reducers are already a part of fuzzer frameworks (Hodován and Kiss, 2018). Our intuition is that the various slices of the fuzzer-generated test case that are investigated during reduction can enrich the spectrum. Thus, in this paper we seek to answer the research question, whether these by-products of reduction can improve SBFL.

The rest of the paper is organized as follows: first, in Section 2, to make this paper self-contained, we

[a] https://orcid.org/0000-0002-8701-5373
[b] https://orcid.org/0000-0002-5072-4774
[c] https://orcid.org/0000-0003-3077-7075

Vince, D., Hodován, R. and Kiss, Á.
Reduction-assisted Fault Localization: Don't Throw Away the By-products!.
DOI: 10.5220/0010560501960206
In *Proceedings of the 16th International Conference on Software Technologies (ICSOFT 2021)*, pages 196-206
ISBN: 978-989-758-523-4

give a brief overview of spectrum-based fault localization and test case reduction. Then, in Section 3, we describe the idea of using reduction by-products in fault localization in detail. In Section 4, we present the results of the experimental evaluation of the idea. In Section 5 we discuss related work, and finally, in Section 6 we summarize our work and conclude the paper.

2 BACKGROUND

Spectrum-based Fault Localization. Given the elements of a program, $|\{e_j\}| = n$, and test 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 ($s_{ij} = 1$) or not ($s_{ij} = 0$). 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 program on test input t_i has resulted in a failure ($r_i = 1$) or not ($r_i = 0$). A typical representation of these two structures is shown below:

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

From these structures, various statistics can be computed for each program element. The most commonly used basic notations are:

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

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.

Various formulae have been proposed to convert these statistics into suspiciousness scores, three of the best-studied (Wong et al., 2016; Pearson et al., 2017) are Tarantula (Jones et al., 2002; Jones and Harrold, 2005), Ochiai (Ochiai, 1957; Abreu et al., 2006; Abreu et al., 2009), and DStar (Wong et al., 2012; Wong et al., 2014), which are computed as follows:

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

$$Ochiai(e) = \frac{c_{ef}(e)}{\sqrt{(c_{ef}(e)+c_{nf}(e)) \cdot (c_{ef}(e)+c_{ep}(e))}}$$

$$D^*(e) = \frac{c_{ef}(e)^*}{c_{nf}(e)+c_{ep}(e)}$$

For all of these formulae, higher scores are assumed to signal more suspicious program elements, i.e., elements that are more likely to contain the fault that is responsible for the test failures. When all program elements are scored, they are ranked. The higher the actually faulty element is ranked, the better the formula.

Test Case Reduction. Given a program with a failure-inducing input, the goal of test case reduction is to produce a smaller test case that still reproduces the failure but is minimal with respect to some definition of minimality. Most techniques (Hildebrandt and Zeller, 2000; Misherghi and Su, 2006; Sun et al., 2018; Gharachorlu and Sumner, 2019) achieve this by iteratively chopping off smaller or larger parts of the input. When such an intermediate test case does not reproduce the failure, it is "thrown away", while failing test cases are trimmed further as long as possible.

The most well-known approach is the minimizing Delta Debugging algorithm (DDMIN) (Zeller, 1999; Hildebrandt and Zeller, 2000; Zeller and Hildebrandt, 2002) that minimizes inputs without information about their format. It works on a set of units representing parts of the test case, e.g., on characters or lines of the input. However, minimizing structured inputs (e.g., program code) with DDMIN can lead to many syntactically incorrect test cases, since DDMIN can break the rules of the input format (e.g., split keywords of a programming language). To help deal with structured inputs, Hierarchical Delta Debugging (HDD) (Misherghi and Su, 2006) uses a tree representation, most often built with the help of a context-free grammar, and applies DDMIN to nodes at every level of the tree.

Test Case Reduction and SBFL. The use of test case reduction in spectrum-based fault localization has been considered by Christi et al. In their study (Christi et al., 2018), they suggested to first reduce failing test cases and then *replace* the failing test cases with their minimized counterparts when performing fault localization. Their results confirmed that SBFL could benefit from the replacement and improve the ranking of the faulty program elements.

3 REDUCTION-ASSISTED FAULT LOCALIZATION

As the experiments of Christi et al. have shown, spectrum-based fault localization can be improved when failing test cases are minimized, and the spectra of the reduced variants are used in statistics and suspiciousness formula calculations instead of the originals (Christi et al., 2018). Our hypothesis is, however, that it is not only the minimized test case that can be helpful, but the by-products – i.e., the intermediate test cases evaluated during reduction – as well. The intuition behind the hypothesis is that during reduction, multiple failing (as well as passing) slices of the original test case are generated. These additional test cases are expected to add extra data to the spectrum matrix and result vector, which may further improve SBFL.

To re-iterate the motivation from Section 1, we assume a fuzzing scenario where there is a test suite that contains passing tests only and a new fuzzer-generated failing test case. We also assume to have a test case reducer that, while trying multiple smaller intermediate variants, produces a reduced version of the original failing test case.

To be able to refer to concepts interesting to the above-described setup, we introduce the following notations. Matrix $\mathbf{S}^{[ts]}$ denotes the spectrum of the test suite ($\{t_i^{[ts]}\}$) that contains passing tests only, i.e., $\mathbf{R}^{[ts]} = \mathbf{0}$. The spectrum of the new failing test case generated by fuzzing (i.e., of $t^{[fz]}$) is represented by matrix $\mathbf{S}^{[fz]}$, consisting only of one row, with the corresponding result vector $\mathbf{R}^{[fz]} = \mathbf{1}$. Reduction outputs a minimal but still failing test case ($t^{[rd]}$), which gives spectrum matrix $\mathbf{S}^{[rd]}$ and result vector $\mathbf{R}^{[rd]} = \mathbf{1}$, also both of one row. The intermediate results, the by-products of reduction, consisting of both failing and passing test cases ($\{t_i^{[by]}\}$) give spectrum matrix $\mathbf{S}^{[by]}$ and result vector $\mathbf{R}^{[by]}$. These can be written as follows.

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

$$\mathbf{S}^{[fz]} = t^{[fz]} \begin{pmatrix} 0/1 & 0/1 & \cdots & 0/1 \end{pmatrix} \quad \mathbf{R}^{[fz]} = \begin{pmatrix} 1 \end{pmatrix}$$

$$\mathbf{S}^{[by]} = \begin{array}{c} t_1^{[by]} \\ t_2^{[by]} \\ \vdots \\ t_p^{[by]} \end{array}\begin{pmatrix} 0/1 & 0/1 & \cdots & 0/1 \\ 0/1 & 0/1 & \cdots & 0/1 \\ \vdots & \vdots & \ddots & \vdots \\ 0/1 & 0/1 & \cdots & 0/1 \end{pmatrix} \mathbf{R}^{[by]} = \begin{pmatrix} 0/1 \\ 0/1 \\ \vdots \\ 0/1 \end{pmatrix}$$

$$\mathbf{S}^{[rd]} = t^{[rd]} \begin{pmatrix} 0/1 & 0/1 & \cdots & 0/1 \end{pmatrix} \mathbf{R}^{[rd]} = \begin{pmatrix} 1 \end{pmatrix}$$

Additionally, we will use the notation $\{t_i^{[by_f]}\}$, $\mathbf{S}^{[by_f]}$, and $\mathbf{R}^{[by_f]} = \mathbf{1}$ to refer to the subset of the by-products, and to their spectrum, which are failing.

These spectra can be combined in various ways to give different inputs to suspiciousness formulae. Combining the spectra of the test suite and the fuzzer-generated failing test gives the information that is usually available to a regular software engineer; we will denote this combination as $\mathbf{S}^{[ts,fz]}$ and $\mathbf{R}^{[ts,fz]}$. The approach suggested by Christi et al. can be formalized as $\mathbf{S}^{[ts,rd]}$ and $\mathbf{R}^{[ts,rd]}$, i.e., as the combination of the spectra (and result vectors) of the test suite and the minimized test case. However, the above discussed spectra allow for further combinations, which are currently in our focus. The by-products of the reduction can also be taken into account during fault localization if $\mathbf{S}^{[ts,by,rd]}$ and $\mathbf{R}^{[ts,by,rd]}$ are used as inputs to the formulae. It may be also worth investigating whether restricting the spectra of the by-products to the failing test cases gives different results, i.e., if $\mathbf{S}^{[ts,by_f,rd]}$ and $\mathbf{R}^{[ts,by_f,rd]}$ are utilized. Finally, the fault localization potential of the reduction stack only may also be of interest – e.g., in cases when no regression test suite is available –, thus we also define $\mathbf{S}^{[fz,by,rd]}$ and $\mathbf{R}^{[fz,by,rd]}$. The above mentioned combinations are shown below.

$$\mathbf{S}^{[ts,fz]} = \begin{bmatrix} \mathbf{S}^{[ts]} \\ \mathbf{S}^{[fz]} \end{bmatrix} \quad \mathbf{R}^{[ts,fz]} = \begin{bmatrix} \mathbf{R}^{[ts]} \\ \mathbf{R}^{[fz]} \end{bmatrix}$$

$$\mathbf{S}^{[ts,rd]} = \begin{bmatrix} \mathbf{S}^{[ts]} \\ \mathbf{S}^{[rd]} \end{bmatrix} \quad \mathbf{R}^{[ts,rd]} = \begin{bmatrix} \mathbf{R}^{[ts]} \\ \mathbf{R}^{[rd]} \end{bmatrix}$$

$$\mathbf{S}^{[ts,by,rd]} = \begin{bmatrix} \mathbf{S}^{[ts]} \\ \mathbf{S}^{[by]} \\ \mathbf{S}^{[rd]} \end{bmatrix} \quad \mathbf{R}^{[ts,by,rd]} = \begin{bmatrix} \mathbf{R}^{[ts]} \\ \mathbf{R}^{[by]} \\ \mathbf{R}^{[rd]} \end{bmatrix}$$

$$\mathbf{S}^{[ts,by_f,rd]} = \begin{bmatrix} \mathbf{S}^{[ts]} \\ \mathbf{S}^{[by_f]} \\ \mathbf{S}^{[rd]} \end{bmatrix} \quad \mathbf{R}^{[ts,by_f,rd]} = \begin{bmatrix} \mathbf{R}^{[ts]} \\ \mathbf{R}^{[by_f]} \\ \mathbf{R}^{[rd]} \end{bmatrix}$$

$$\mathbf{S}^{[fz,by,rd]} = \begin{bmatrix} \mathbf{S}^{[fz]} \\ \mathbf{S}^{[by]} \\ \mathbf{S}^{[rd]} \end{bmatrix} \quad \mathbf{R}^{[fz,by,rd]} = \begin{bmatrix} \mathbf{R}^{[fz]} \\ \mathbf{R}^{[by]} \\ \mathbf{R}^{[rd]} \end{bmatrix}$$

It shall be noted that the above described idea of using the spectra of the by-products of test case reduction can be generalized to use-cases when there are multiple failing test cases – i.e., when $\mathbf{S}^{[fz]}$, and therefore $\mathbf{S}^{[rd]}$ too, have multiple rows – or when the failing test was not found by fuzzing but was already part of the test suite. The generalization to the first case is trivial: $\mathbf{S}^{[fz]}$ and $\mathbf{S}^{[rd]}$ having multiple rows has no effect on the basic concepts, and $\mathbf{S}^{[by]}$ shall simply contain the by-products of the reduction of all failing test cases. The generalization to the second case is also straightforward: the passing tests of the test suite will constitute $\{t_i^{[\mathrm{ts}]}\}$, while the failing test of the test suite becomes $t^{[\mathrm{fz}]}$.

4 EXPERIMENTAL RESULTS

Experiment Setup. To evaluate the idea of using the by-products of test case reduction in fault localization, we had to look for collections of reducible test cases first; and we have settled with two such projects. The first of them is the JerryScript Reduction Test Suite (JRTS)[1] with the underlying JerryScript lightweight JavaScript engine[2]. JRTS contains fuzzer-generated test inputs (i.e., JavaScript sources) that trigger bugs (e.g., assertion failures or memory corruptions) in various versions of JerryScript. All the bugs have already been reported to the issue tracker of the engine with minimized test inputs (and got fixed), but JRTS contains the original test cases as they were first found by a fuzzer. For every test case, JRTS also records the version of JerryScript that exhibits the bug (that was not captured by its regression test suite at that version) and contains a test oracle that determines the outcome of a test input as failing or passing based on whether or not it reproduces the same failure as the original test case. We have used this test suite because it perfectly aligns with the scenario envisioned in Section 3.

The second set of inputs comes from the Siemens/SIR suite[3] (Hutchins et al., 1994; Do et al., 2005), used by many to evaluate SBFL techniques (Harrold et al., 2000; Christi et al., 2018). The suite contains multiple versions of programs, an original correct variant and several others with seeded faults for each, as well as a test suite per program. Each faulty program version causes multiple test cases to fail in the corresponding test suite. Originally, the tests in the suite determine their outcome by comparing actual and precomputed expected outputs for a given input. However, this makes reducing failing inputs non-trivial, because the suite does not contain the expected outputs for the new test cases generated during reduction. To solve this problem, our modified test oracles utilize the original program versions to generate the expected output for every test input and compare that to the output of the faulty program versions.

[1]https://github.com/vincedani/jrts

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

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

To minimize failing test cases, we have used multiple test case reducers. For JRTS, we have used the HDD-based Picireny tool[4] with the JavaScript grammar from the ANTLR v4 grammars repository[5] to build the tree representation of the inputs. (For the sake of reproducibility, we mention that before performing HDD, squeezing of linear components (Hodován et al., 2017b), and flattening of recursive structures (Hodován et al., 2017a) have been applied to the trees, and DDMIN within HDD was configured to skip subset tests, perform complement tests in backward syntactic order (Hodován and Kiss, 2016), and use content caching (Hodován et al., 2017b).) As the format of the inputs in the Siemens/SIR suite is unstructured or unknown, we have used the Picire[6] implementation of the structure-unaware DDMIN algorithm for their reduction. The reducer was configured to use character granularity in most of the cases, except for the inputs of the *tot_info* application where line-based reduction was applied, like in (Christi et al., 2018).

To obtain the program element hit spectra, we have compiled all applications (i.e., the JerryScript engine as well as the programs of the Siemens/SIR suite) with instrumentation for coverage analysis, and gathered function-level coverage information after the execution of every test input using the LCOV[7] tool. (According to several sources, function-level granularity is suitable for SBFL purposes (Kochhar et al., 2016; B. Le et al., 2016; Beszédes et al., 2020).) Note that although the Siemens/SIR suite contains precomputed coverage information for every test case of every program version, it naturally does not contain coverage information for the reduced test cases or for the by-products of the reduction. Thus, to ensure consistent results, we have used the LCOV-based coverage information collection approach for all test cases.

The experiments were executed on a workstation equipped with an Intel Core i5-9400 CPU clocked at 2.9 GHz and 16 GB RAM. The machine was running Ubuntu 20.04 with Linux kernel 5.4.0.

Results. Table 1 shows the size of the spectra collected on JRTS. The *Issue* column indicates the ID assigned to the bug report in the JerryScript project repository that corresponds to the test case. The *Functions* column shows the total number of functions in the version of the engine specific to the issue. The numbers of executed regression tests, fuzzed test cases, by-products of reduction, and reduced

[4]https://github.com/renatahodovan/picireny

[5]https://github.com/antlr/grammars-v4

[6]https://github.com/renatahodovan/picire

[7]https://github.com/linux-test-project/lcov

Table 1: Size of spectra collected on the JerryScript Reduction Test Suite.

Issue	**Functions**	**Tests from Suite** $c_{pass}^{[ts]}$	**Test from Fuzzing** $c_{fail}^{[fz]}$	**By-products of Reduction** $c_{pass}^{[by]} + c_{fail}^{[by]}$	**Test from Reduction** $c_{fail}^{[rd]}$
#3361	1,511	2,144	1	129 + 15	1
#3376	1,519	2,152	1	99 + 20	1
#3431	1,539	2,174	1	44 + 9	1
#3433	1,537	2,178	1	11 + 7	1
#3437	1,548	2,181	1	35 + 14	1
#3479	1,586	2,199	1	196 + 37	1
#3483	1,586	2,200	1	61 + 8	1
#3506	1,587	2,207	1	97 + 18	1
#3523	1,606	2,222	1	99 + 12	1
#3534	1,608	2,227	1	160 + 13	1
#3536	1,608	2,228	1	139 + 11	1

Table 2: Size of spectra collected on the Siemens/SIR suite.

Program	**Functions**	**Passing Tests from Suite**† $c_{pass}^{[ts]}$	**Failing Tests from Suite**† $c_{fail}^{[fz]}$	**By-products of Reduction**† $c_{pass}^{[by]} + c_{fail}^{[by]}$	**Tests from Reduction**† $c_{fail}^{[rd]}$
print_tokens	18	28,450	530	8,692 + 4,721	530
print_tokens2	19	38,957	2,443	18,670 + 15,768	2,443
replace	21	121,510	2,374	9,148 + 10,847	2,374
schedule	18	15,637	4,358	15,180 + 32,782	4,358
schedule2	16	22,267	201	6,484 + 2,158	201
tot_info	7	22,861	1,979	58,063 + 11,760	1,979

†Sum of test counts from all fault-seeded program versions.

test cases are in columns *Tests from Suite*, *Test from Fuzzing*, *By-products of Reduction*, and *Test from Reduction*, respectively. (For the by-products of reduction, the number of the passing and failing test cases are shown separately.)

Table 2 shows the same information for the Siemens/SIR suite. As mentioned above, the tests were not found by fuzzing in this case, but they were part of the original suite. This is also reflected in the names of the columns *Passing Tests from Suite* and *Failing Tests from Suite*; but to keep the notations consistent throughout the paper, we keep referring to these values as $c_{pass}^{[ts]}$ and $c_{fail}^{[fz]}$, respectively, as also discussed in Section 3. The suite contains multiple faulty versions of each program and every fault is detected by multiple test cases, therefore the numbers of tests (both passing and failing from suite, the by-products, and from reduction) show summed values across all versions.

Using the above spectra and using their combinations as discussed in Section 3, we have computed the Tarantula, Ochiai, and D^2 suspiciousness scores for every function of every JerryScript version and every faulty Siemens/SIR program[8,9]. In Tables 3 and 4, we show the average rank of the faulty functions, which have been manually identified in the bug fixing patches of JerryScript and in the original-vs-fault-seeded program source diffs of the Siemens/SIR suite. (Average rank means the use of fractional or “1 2.5 2.5 4” ranking, i.e., when multiple functions get the same suspiciousness score, they all receive the same rank, which is the mean of what they would get under distinct ordinal ranking.) We use $\mathrm{rk}(e)$ to denote the computed (average) rank of a program element with a superscript to signal the spectrum combination used for the ranking, and we use f^* to denote the manually identified faulty function. Thus, we have the following values in the tables:

- $\mathrm{rk}^{[ts,rd]}(f^*)$: The rank of the faulty function computed using the combination of the spectra of the

[8]Division by zero may occur during the computation of all three scores. We have chosen to define division by zero as zero in the Tarantula and Ochiai formulae, and as a suitably large number ($c_{ef}(e)^* + 1$) in D*. A detailed discussion of this issue is given in the Appendix.

[9]The parameter of the D* formula can be freely chosen, but $* = 2$ is the most thoroughly explored configuration (Pearson et al., 2017), thus we have also used this value.

Table 3: Average rank of faulty functions in JRTS.

Issue	**Formula**	$\text{rk}^{[\text{ts,rd}]}(f^*)$	$\text{rk}^{[\text{ts,by,rd}]}(f^*)$	$\text{rk}^{[\text{ts,by}_f\text{,rd}]}(f^*)$	$\text{rk}^{[\text{fz,by,rd}]}(f^*)$
#3361	Tarantula	4	***3***	***3***	88
	Ochiai	4	***3***	***3***	20
	D^2	4	***3***	***3***	19
#3376	Tarantula	39.5	***27***	*29*	72
	Ochiai	39.5	*18*	*25.5*	***12***
	D^2	39.5	*18*	*25.5*	***12***
#3431	Tarantula	276	***145.5***	*188.5*	*271.5*
	Ochiai	276	***135***	*178*	*223.5*
	D^2	276	***134***	*176*	*216.5*
#3433	Tarantula	10	***5.5***	*7*	*7*
	Ochiai	10	***5***	*6.5*	***5***
	D^2	10	***5***	*6.5*	***5***
#3437	Tarantula	260	***157.5***	*200*	377.5
	Ochiai	260	***130.5***	*175*	*194.5*
	D^2	260	***129***	*172*	*193.5*
#3479	Tarantula	4	***2***	*2.5*	20.5
	Ochiai	4	***2***	***2***	6.5
	D^2	4	***2***	***2***	6.5
#3483	Tarantula	192.5	***111***	*154*	229
	Ochiai	192.5	***102.5***	*145*	*183*
	D^2	192.5	***101***	*144*	*162*
#3506	Tarantula	5	***4.5***	***4.5***	61.5
	Ochiai	5	***2.5***	*3*	9.5
	D^2	5	***2.5***	*3*	9.5
#3523	Tarantula	**7**	8.5	8.5	56
	Ochiai	7	***5.5***	*6*	16
	D^2	7	***4***	*4.5*	16
#3534	Tarantula	20	***3***	*10.5*	73.5
	Ochiai	20	***3***	*10.5*	*16.5*
	D^2	20	***3***	*10.5*	*15.5*
#3536	Tarantula	3	***2***	***2***	8
	Ochiai	3	3	***2***	6.5
	D^2	3	***2***	***2***	6.5

test suite and that of the reduced test case. We will use this value as our baseline. (Note that we do not list $\text{rk}^{[\text{ts,fz}]}(f^*)$ nor use it as a baseline, as the work of Christi et al. has already shown $\text{rk}^{[\text{ts,rd}]}(f^*)$ to be better.)

- $\text{rk}^{[\text{ts,by,rd}]}(f^*)$: The rank of the faulty function computed with the assistance of reduction, i.e., using all of the by-products of reduction as well.
- $\text{rk}^{[\text{ts,by}_f\text{,rd}]}(f^*)$: The same as above, but using the failing by-products only.
- $\text{rk}^{[\text{fz,by,rd}]}(f^*)$: The rank of the faulty function computed without the spectrum of the test suite, but with the spectra of the reduction stack only, i.e., based on those of the fuzzer-generated test case, the by-products of reduction, and the minimized test case.

In every row of the tables, numbers in italics denote ranks better than the baseline, while bold numbers denote the best rank(s). (Note that the smaller the numerical values the better, i.e., the highest and best possible rank is 1).

The results measured on JRTS (shown in Table 3) show that the by-products of reduction (both with and without the passing test cases) helped improve fault localization. Both $\text{rk}^{[\text{ts,by,rd}]}(f^*) < \text{rk}^{[\text{ts,rd}]}(f^*)$ and $\text{rk}^{[\text{ts,by}_f\text{,rd}]}(f^*) < \text{rk}^{[\text{ts,rd}]}(f^*)$ hold for almost all issues and suspiciousness formulae. The only two exceptions are the ranking based on the Tarantula scores for issue #3523, where the rank of the faulty function became slightly worse (lowered from 7 to 8.5), and the ranking based on the Ochiai formula for issue #3536, where the rank of the faulty function did not change

Table 4: Average rank of faulty functions in the Siemens/SIR suite.

Program	**Formula**	$\mathrm{rk}^{[\mathrm{ts,rd}]}(f^*)^{‡}$	$\mathrm{rk}^{[\mathrm{ts,by,rd}]}(f^*)^{‡}$	$\mathrm{rk}^{[\mathrm{ts,by}_f\mathrm{,rd}]}(f^*)^{‡}$	$\mathrm{rk}^{[\mathrm{fz,by,rd}]}(f^*)^{‡}$
print_tokens	Tarantula	**5.2**	5.5	5.4	7.1
	Ochiai	**5.2**	**5.2**	**5.2**	**5.2**
	D^2	**5.2**	**5.2**	**5.2**	**5.2**
print_tokens2	Tarantula	**6.75**	7.05	7.2	7.6
	Ochiai	6.65	***6.55***	6.75	7.1
	D^2	6.65	***6.55***	6.7	7.1
replace	Tarantula	**3.57**	**3.57**	3.59	8.78
	Ochiai	3.04	***3***	*3.02*	6.39
	D^2	3.04	***3***	*3.02*	6.31
schedule	Tarantula	8.94	*8.63*	***7.88***	8.94
	Ochiai	8.88	*8.31*	***8***	*8.81*
	D^2	8.88	*8.25*	***8.13***	*8.44*
schedule2	Tarantula	**4.28**	4.56	4.44	7.22
	Ochiai	**4**	4.22	4.11	5.44
	D^2	**4**	4.22	4.11	5.44
tot_info	Tarantula	**1.53**	**1.53**	1.65	1.78
	Ochiai	**1.45**	1.48	**1.45**	1.48
	D^2	**1.47**	1.48	**1.47**	1.5

‡Averaged over all fault-seeded program versions.

when the spectra of all passing by-products were considered. However, if we focus on the D^2 formula only, then strict improvement can be observed for all issues. (Also note that for all test cases of JRTS, D^2 performs at least as well as the other two formulae).

On average, the improvement of $\mathrm{rk}^{[ts,by,rd]}(f^*)$ *over* $\mathrm{rk}^{[ts,rd]}(f^*)$ *is 35.24%, 47%, and 49.1% with the Tarantula, Ochiai, and D^2 formulae, respectively. For* $\mathrm{rk}^{[ts,by_f,rd]}(f^*)$*, the average improvement over* $\mathrm{rk}^{[ts,rd]}(f^*)$ *is 23.93%, 33.95%, and 36.11% with Tarantula, Ochiai, and D^2, respectively. I.e., even the failing by-products of reduction helped improve fault localization, but keeping the passing by-products as well yielded even better results.*

When the spectrum of the regression test suite is not used for fault localization – i.e., only the fuzzer-generated test input, the by-products of reduction, and the minimized test case contribute to the spectrum – , then the results are mixed. Using this spectrum as input, even the D^2 formula ranked the faulty functions lower than with the baseline spectrum for 5 of the 11 issues, and with Tarantula, this was the case for 9 of the 11 issues. Thus, this restricted set of test cases should only be used for fault localization when there really is no other test suite available.

When it comes to the data of Table 3, three rows deserve additional discussion: the rankings at issues #3431, #3437, and #3483. For these issues, $\mathrm{rk}^{[\mathrm{ts,rd}]}(f^*)$ falls in the range of hundreds with all formulae. In these cases, the actual bug is far away from the point in the engine where the fault is eventually manifested, which seems to mislead the suspiciousness formulae. Although our proposal to use the by-products of reduction did not fix this problem entirely, the ranks have improved considerably, e.g., from 276 to 134, from 260 to 129, and from 192.5 to 101 when using D^2 on $\mathbf{S}^{[\mathrm{ts,by,rd}]}$.

The results measured on the Siemens/SIR suite (shown in Table 4) are somewhat less significant. The ranks of the faulty functions (averaged over the fault-seeded versions for each program) do not change prominently with any of the spectrum combinations or suspiciousness formulae. In general, the ranks of the faulty functions computed with Tarantula or using the spectrum of the reduction stack only (i.e., $\mathbf{S}^{[\mathrm{fz,by,rd}]}$) became lower, but not by orders of magnitude. With Ochiai and D^2, the ranks improved on average, but also only by a small factor (by less than 1%).

Based on the data and observations above, we can conclude that adding the by-products of reduction to the minimized test case and to the existing regression test suite can improve the localization of faults revealed by fuzzer-generated test cases, especially with the Ochiai and D^* $(* = 2)$ formulae.

5 RELATED WORK

The closest to our work is the study of Christi et al., where Delta Debugging-based test case reduction was also suggested to improve fault localization (Christi et al., 2018). Their approach was to use the reduced test case for fault localization instead of the fuzzer-generated one, assuming that the minimal test case that reproduces the original failure contains less misleading information. Xuan and Monperrus also proposed test case purification in order to improve fault localization (Xuan and Monperrus, 2014). Their goal was to generate purified – i.e., minimized – versions of unit tests that included only one assertion and excluded statements unrelated to the assertion. They used an automated test case generator to produce single-assertion test cases for each failed unit test, then applied slicing to remove code parts unrelated to that assertion. Both of these works focused on the minimized-purified test cases, but did not take the by-products of reduction into consideration.

Several studies have been carried out about using test *suite* reduction to improve fault localization. Vidács et al. investigated different test suite reduction approaches from performance and detection points of view, and proposed a combined method which incorporated both aspects (Vidács et al., 2014). Fu et al. proposed a similarity-based test suite reduction approach (Fu et al., 2017) to extract highly suspicious statements and select similar passing test cases for each failing one. We see these techniques – and test suite reduction in general – as orthogonal to our approach, and their combination may be worth investigating in future research.

In a wider sense, there are a great number of works related to the topic of this paper. Both of the two research areas that are interconnected in this paper – i.e., spectrum-based fault localization and test case reduction – have huge literatures on their own. Therefore, we refer the reader to recent surveys and overviews of the two research areas for further information (Wong et al., 2016; Zeller, 2021).

6 SUMMARY

In this paper, we have proposed to utilize test case reduction to assist spectrum-based fault localization in a fuzzing-motivated scenario. When an application is being fuzz-tested, it is typical that when a failure is observed, there is only a single test input that triggers that failure – i.e., the test case randomly generated by a fuzzer – while all other already existing tests pass. Such heavily unbalanced results can pose problems to spectrum-based fault localization techniques. Test case reduction is a technique that is already commonly used together with fuzz testing to minimize the otherwise unnecessarily large randomly generated test cases. Strictly speaking, for test case reduction, the only valuable output is the minimized test case. However, the intermediate results, or by-products, of the reduction are a mix of additional failing and passing test cases to the tested application. Therefore, we have proposed to use these by-products as well when applying SBFL to locate the fault. We have evaluated this idea, and our experimental results show that the extension of the existing test suite with the failing and passing by-products of test case reduction can help SBFL, i.e., the rank of the faulty program element (function) can improve by up to 49% on a real-world use-case. The experimental results also show that the here-proposed idea is not specific to a given SBFL formula, as improvements have been measured with three widely used formulae (Tarantula, Ochiai, and D^2).

We see several potential future directions to continue this research. We are interested in how different test case reduction techniques can assist or affect fault localization – e.g., variants of DDMIN or HDD, like HDDr or Coarse HDD, or techniques that are not H/DD-based, e.g., Perses or Pardis. We plan to extend the current experiment to see how reduction-assisted fault localization scales to different granularities, e.g., to statement-level fault localization. We would like to validate our results on a wider set of subjects, e.g., on programs with different input formats and written in different programming languages. Finally, we also wish to investigate the interplay between reduction-assisted fault localization and test suite reduction.

ACKNOWLEDGEMENTS

This research was supported by the EU-supported Hungarian national grant GINOP-2.3.2-15-2016-00037 and by grant NKFIH-1279-2/2020 of the Ministry for Innovation and Technology, Hungary.

REFERENCES

Abreu, R., Zoeteweij, P., Golsteijn, R., and van Gemund, A. J. C. (2009). A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792.

Abreu, R., Zoeteweij, P., and van Gemund, A. J. C. (2006). An evaluation of similarity coefficients for software fault localization. In *Proceedings of the 12th Pacific*

Rim International Symposium on Dependable Computing (PRDC), pages 39–46. IEEE.

B. Le, T.-D., Lo, D., Le Goues, C., and Grunske, L. (2016). A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA)*, pages 177–188. ACM.

Beszédes, Á., Horváth, F., Di Penta, M., and Gyimóthy, T. (2020). Leveraging contextual information from function call chains to improve fault localization. In *Proceedings of the 27th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 468–479. IEEE.

Cheetham, A. H. and Hazel, J. E. (1969). Binary (presence-absence) similarity coefficients. *Journal of Paleontology*, 43(5):1130–1136.

Christi, A., Olson, M. L., Alipour, M. A., and Groce, A. (2018). Reduce before you localize: Delta-debugging and spectrum-based fault localization. In *Proceedings of the 2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 184–191. IEEE.

Do, H., Elbaum, S., and Rothermel, G. (2005). Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435.

Fu, W., Yu, H., Fan, G., Ji, X., and Pei, X. (2017). A test suite reduction approach to improving the effectiveness of fault localization. In *Proceedings of the 2017 International Conference on Software Analysis, Testing and Evolution (SATE)*, pages 10–19.

Gharachorlu, G. and Sumner, N. (2019). PARDIS: Priority aware test case reduction. In *Fundamental Approaches to Software Engineering – 22nd International Conference, FASE 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, volume 11424 of *Lecture Notes in Computer Science (LNCS)*, pages 409–426. Springer.

Harrold, M. J., Rothermel, G., Sayre, K., Wu, R., and Yi, L. (2000). An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification and Reliability*, 10(3):171–194.

Hildebrandt, R. and Zeller, A. (2000). Simplifying failure-inducing input. In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pages 135–145. ACM.

Hodován, R. and Kiss, Á. (2016). Practical improvements to the minimizing delta debugging algorithm. In *Proceedings of the 11th International Joint Conference on Software Technologies (ICSOFT) – Volume 1: ICSOFT-EA*, pages 241–248. SciTePress.

Hodován, R. and Kiss, Á. (2018). Fuzzinator: An open-source modular random testing framework. In *Proceedings of the 11th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 416–421. IEEE.

Hodován, R., Kiss, Á., and Gyimóthy, T. (2017a). Coarse hierarchical delta debugging. In *Proceedings of the 33rd IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 194–203. IEEE.

Hodován, R., Kiss, Á., and Gyimóthy, T. (2017b). Tree preprocessing and test outcome caching for efficient hierarchical delta debugging. In *Proceedings of the 12th IEEE/ACM International Workshop on Automation of Software Testing (AST)*, pages 23–29. IEEE.

Hutchins, M., Foster, H., Goradia, T., and Ostrand, T. (1994). Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering (ICSE)*, pages 191–200. IEEE.

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

Jones, J. A., Harrold, M. J., and Stasko, J. (2002). Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering (ICSE)*, pages 467–477. ACM.

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

Landsberg, D., Chockler, H., Kroening, D., and Lewis, M. (2015). 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*, volume 9033 of *Lecture Notes in Computer Science (LNCS)*, pages 115–129. Springer.

Lee, H. J. (2011). *Software debugging using program spectra*. PhD thesis, Department of Computer Science and Software Engineering, The University of Melbourne.

Misherghi, G. and Su, Z. (2006). HDD: Hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pages 142–151. ACM.

Naish, L. and Lee, H. J. (2013). Duals in spectral fault localization. In *Proceedings of the 2013 22nd Australian Software Engineering Conference (ASWEC)*, pages 51–59. IEEE.

Naish, L., Lee, H. J., and Ramamohanarao, K. (2011). A model for spectra-based software diagnosis. *ACM Transactions on Software Engineering and Methodology*, 20(3):11:1–11:32.

Naish, L., Neelofar, and Ramamohanarao, K. (2015). Multiple bug spectral fault localization using genetic programming. In *Proceedings of the 24th Australasian Software Engineering Conference (ASWEC)*, pages 11–17. IEEE.

Ochiai, A. (1957). Zoogeographical studies on the soleoid fishes found in Japan and its neighhouring regions–II.

Bulletin of the Japanese Society of Scientific Fisheries, 22(9):526–530.

Pearson, S., Campos, J., Just, R., Fraser, G., Abreu, R., Ernst, M. D., Pang, D., and Keller, B. (2017). Evaluating and improving fault localization. In *Proceedings of the 39th IEEE/ACM International Conference on Software Engineering (ICSE)*, pages 609–620. IEEE.

Reps, T., Ball, T., Das, M., and Larus, J. (1997). The use of program profiling for software maintenance with applications to the year 2000 problem. *ACM SIGSOFT Software Engineering Notes*, 22(6):432–449.

Sokal, R. R. and Sneath, P. H. A. (1963). *Principles of Numerical Taxonomy*. W. H. Freeman and Company.

Sun, C., Li, Y., Zhang, Q., Gu, T., and Su, Z. (2018). Perses: Syntax-guided program reduction. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, pages 361–371. ACM.

Takanen, A., DeMott, J., Miller, C., and Kettunen, A. (2018). *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, 2nd edition.

Troya, J., Segura, S., Parejo, J. A., and Ruiz-Cortés, A. (2018). Spectrum-based fault localization in model transformations. *ACM Transactions on Software Engineering and Methodology*, 27(3):13:1–13:50.

Vidács, L., Beszédes, Á., Tengeri, D., Siket, I., and Gyimóthy, T. (2014). Test suite reduction for fault detection and localization: A combined approach. In *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pages 204–213. IEEE.

Wong, W. E., Debroy, V., Gao, R., and Li, Y. (2014). The DStar method for effective software fault localization. *IEEE Transactions on Reliability*, 63(1):290–308.

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

Wong, W. E., Gao, R., Li, Y., Abreu, R., and Wotawa, F. (2016). A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740.

Xuan, J. and Monperrus, M. (2014). Test case purification for improving fault localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 52–63. ACM.

Xue, X. and Namin, A. S. (2013). 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)*, pages 113–122. IEEE.

Yoo, S. (2012). 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*, volume 7515 of *Lecture Notes in Computer Science (LNCS)*, pages 244–258. Springer.

Zeller, A. (1999). Yesterday, my program worked. Today, it does not. Why? In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, volume 1687 of *Lecture Notes in Computer Science*, pages 253–267. Springer.

Zeller, A. (2021). Reducing failure-inducing inputs. In *The Debugging Book*. CISPA Helmholtz Center for Information Security. https://www.debuggingbook.org/html/DeltaDebugger.html [Retrieved 2021-04-06].

Zeller, A. and Hildebrandt, R. (2002). Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200.

APPENDIX

Several SBFL formulae contain divisions, and several of them are not well-defined for all possible inputs, as their computation may involve divisions by zero. Multiple approaches exist in the literature to deal with such cases, mostly defining a variant of division that is defined for a zero denominator or by modifying the values used in the formulae.

Approaches in the Literature: We quote eight papers from the literature of the past two decades that discussed this topic, and suggested and used different approaches.

(Jones and Harrold, 2005, p. 274): *"Note that if any of the denominators evaluate to zero, we assign zero to that fraction."*

(Naish et al., 2011, p. 5): *"Several of the metrics contain quotients where the denominator can be zero. If the numerator is zero we use zero otherwise we use a suitably large value. For example, the Overlap formula we can use the number of tests plus 1, which is larger than any value which can be returned with a non-zero denominator. An alternative is to add a suitably small ε to the denominator."*

(Lee, 2011, p. 73): *"When it comes to ranking program statements, there is a possibility of the denominator of respective spectra metrics having zero. We could handle this scenario in three different ways.*

1. *Return a large metric value*
2. *Assign zero to the statement*
3. *Use ε on the denominator*

[...] For example, when using the Tarantula metric to evaluate the metric value of program statements, if the denominator of a statement is zero, rather than returning an undefined value, we could use a larger value such as the number of tests plus 1, which is larger than any value which can be returned with a

non-zero denominator. [...] The third solution proposed to handle the denominator being zero is to add a suitably small ε to the denominator. There is no issue when applying ε on the denominator for most of the spectra metrics with the exception of the Ample metric."

(Yoo, 2012, p. 249): *"The division operator* `gp_div` *will return 1 when division by zero error is expected. [...]* `gp_div(a, b)` *1 if $b = 0$, $\frac{a}{b}$ otherwise"*

(Naish and Lee, 2013, p. 56): *"As well as the new metrics, the table includes the original version of Jaccard and its duals and the original version of Tarantula, modified to avoid division by zero ($x/0$ is considered to be 0.5 if $x = 0$ and 9999 otherwise)."*

(Xue and Namin, 2013, p. 115): *"where the constant 0.1 is added to avoid division by zero and computational problems as suggested by Liu and Motoda"*

(Landsberg et al., 2015, p. 124): *"To assign a score, we added a small* prior constant *(0.5) to each cell of each program entity's contingency table in order to avoid divisions by zero, as is convention (Naish et al., 2011)."*[10]

(Troya et al., 2018, p. 22): *"Different approaches mention how to deal with such cases (Naish et al., 2015; Xue and Namin, 2013; Yoo, 2012). Following the guidelines of these works, if a denominator is zero and the numerator is also zero, then our computation returns zero. However, if the numerator is not 0, then it returns 1 (Yoo, 2012)."*[11]

These approaches can be formalized and grouped either as modifying division like

- $\mathrm{div}^{\langle 0,0\rangle}$ (Jones and Harrold, 2005; Lee, 2011),
- $\mathrm{div}^{\langle 0,N\rangle}$ (Naish et al., 2011),
- $\mathrm{div}^{\langle N,N\rangle}$ (Lee, 2011),
- $\mathrm{div}^{\langle 1,1\rangle}$ (Yoo, 2012),
- $\mathrm{div}^{\langle 0.5,9999\rangle}$ (Naish and Lee, 2013),
- $\mathrm{div}^{\langle 0,1\rangle}$ (Troya et al., 2018),
- $\mathrm{div}_{\langle +\varepsilon\rangle}$ (Naish et al., 2011; Lee, 2011),

or as modifying the values of the c_{ef}, c_{nf}, c_{ep}, and c_{np} like

- $c + \varepsilon$ (Xue and Namin, 2013; Landsberg et al., 2015),

where

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

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

and N and ε are suitably large and small numbers, respectively.

The Approach used in This Paper: We argue, however, that there may be no single solution applicable to all formulae. If their authors have not explicitly defined how to deal with a zero denominator, then each formula should be analyzed and augmented individually. (But if the authors did define the interpretation of a division by zero, then their definition should be followed.)

Several suspiciousness formulae used for SBFL originate from the domain of systematic biological research (where they are called coefficients) and have been developed decades (some even a century) ago. Thus, fortunately, many of them have already been thoroughly analyzed. So it has been shown (Cheetham and Hazel, 1969) that the Ochiai coefficient (Ochiai, 1957) – which is exactly equivalent to the Ochiai formula (Abreu et al., 2006) – tends to zero as the denominator tends to zero. The D^* formula is a relatively new construct (Wong et al., 2012), but it is actually a modified version of the 1st Kulczynski coefficient (Sokal and Sneath, 1963), which has been shown to tend to infinity as the denominator tends to zero. Therefore, we decided to assign zero to the Ochiai formula and infinity (approximated with a suitably large number) to D^* when their denominator is zero. I.e., we have used $\mathrm{div}^{\langle 0,0\rangle}$ in Ochiai and $\mathrm{div}^{\langle N,N\rangle}$ in D^*.

However, we did not perform any analyses on the Tarantula formula because its authors were explicit about interpreting all division-by-zeros as zero (Jones and Harrold, 2005). Thus, we have followed their definition and used $\mathrm{div}^{\langle 0,0\rangle}$ in that formula.

[10]Incorrectly cites (Naish et al., 2011), which does not mention anything like that. Might have wanted to cite (Naish and Lee, 2013), which mentions a constant 0.5, but not to be added to each cell of the contingency table but to be used as the result of 0/0.

[11](Naish et al., 2015) does not mention explicitly how to deal with division by zero, but references (Landsberg et al., 2015) as related work.