Augmenting Automated Spectrum Based Fault Localization For Multiple Faults

Prantik Chatterjee[1] , **José Campos**[2,3] , **Rui Abreu**[2,4] and **Subhajit Roy**[1]
[1]Indian Institute of Technology Kanpur, India
[2]Faculty of Engineering of University of Porto, Porto, Portugal
[3]LASIGE, Faculdade de Ciências, Universidade de Lisboa, Lisboa, Portugal
[4]INESC-ID, Porto, Portugal
{prantik, subhajit}@cse.iitk.ac.in, jcmc@fe.up.pt, rui@computer.org

Abstract

Spectrum-based Fault Localization (SBFL) uses the coverage of test cases and their outcome (pass/fail) to predict the "suspiciousness" of program components, e.g., lines of code. SBFL is, perhaps, the most successful fault localization technique due to its simplicity and scalability. However, SBFL heuristics do not perform well in scenarios where a program may have multiple faulty components. In this work, we propose a new algorithm that "augments" previously proposed SBFL heuristics to produce a ranked list where faulty components ranked low by base SBFL metrics are ranked significantly higher. We implement our ideas in a tool, ARTEMIS, that attempts to "bubble up" faulty components which are ranked lower by base SBFL metrics. We compare our technique to the most popular SBFL metrics and demonstrate statistically significant improvement in the developer effort for fault localization with respect to the basic strategies.

1 Introduction

As softwares continue to become larger and more complex, testing and debugging have become considerably expensive. Naturally, the domain of automated testing and debugging of softwares has attracted quite a lot of attention in the past decade. Spectrum-based Fault Localization (SBFL) techniques have proven to be very popular due to their simplicity and scalability. SBFL techniques attempt to identify the faulty components in a program based on a statistical analysis of the test *spectra*, that captures information on the *activity pattern* of each test in the test-suite (which test executes which components) and the *resulting outcomes* (which tests fail).

The tests can be either manually written by developers or generated automatically by evolutionary search based software testing tools such as EVOSUITE [Fraser and Arcuri, 2011] which employ fitness functions targeted towards fault diagnosis (such as ULYSIS [Chatterjee *et al.*, 2020], DDU [Perez *et al.*, 2017]) to search for optimal test suites. Automated testing has been found to be very effective in providing more coverage and exposing unexpected faults in programs [Serra *et al.*, 2019]. This work is directed towards improving the automated fault localization pipeline, which employs intelligent search strategies to find optimal test suites and localize suspicious program components which are most likely to be buggy.

While the SBFL metrics are potent when the culprit of observed failures lies in a single location in the program, they tend to be less effective when the program contains multiple faulty components [Wong *et al.*, 2016]. We identified that a primary reason for this is the presence of *dominating* faulty components: the number of failures due a few faulty components overwhelm the others.

In this paper, we use an approach to counter this challenge via a *multiverse* approach: we consider multiple universes, each universe assuming that one of the top-ranked components is faulty. We compute a re-ranking of the components in each of the universes by *simulating* the case that the faulty component is fixed. Simulating a repair of failures caused by the dominating fault "bubbles up" the ranks for the other faulty components. This continues recursively till all failing tests are assumed to be fixed. Each list collected across the universes correspond to different faults. We provide an algorithm to merge the lists over the multiverse into one list where the dormant faults are "bubbled" up and can therefore be identified with lesser effort.

We build our ideas into a tool, ARTEMIS that generates an augmented ranked list of components based on the program spectrums. ARTEMIS is parametric on SBFL metrics; any SBFL metric can be "plugged-in". We perform experiments on six most popular SBFL metrics, viz. Ochiai [Abreu *et al.*, 2009a], Tarantula [Jones and Harrold, 2005], Barinel [Abreu *et al.*, 2009c], Op2 [Naish *et al.*, 2011], Dstar [Wong *et al.*, 2013] and Kulczynski [Choi *et al.*, 2010]. We select these metrics due to recent studies that cite their effectiveness over others [Pearson *et al.*, 2017].

Our experiments show that ARTEMIS provides a statistically significant improvement of approximately 17% mean and 14% median over the base SBFL metrics in terms of developer effort ($\mathcal{EXAM}$ score) on average. The primary contributions of this work are as follows:

- We design a novel multiverse analysis approach to identify "dormant" faults which are ranked low in a multi-fault scenario.

- We build our ideas in a tool called ARTEMIS.
- We show that ARTEMIS generalizes over unknown data.
- We compare ARTEMIS against the most popular SBFL metrics and show that ARTEMIS outperforms all of them.

We provide an appendix with additional experimental results and the source code of ARTEMIS *in https://github.com/prantikchatterjee/ArtemisIJCAI23.git*

2 Background

We provide a more in-depth overview on SBFL in this section. Readers familiar with SBFL may want to skip this section.

Spectrum-based Fault Localization (SBFL) works by executing a program on a diverse set of inputs (a test-suite) and collecting the behavior of the program in a program spectrum. A spectrum consists of two elements: an activity matrix and an error vector. An activity matrix A is a binary matrix where each row corresponds to a test-case and records which components were executed in this test-case. If the test-suite consists of n test-cases and the program under consideration has m components, then the dimension of A is $n \times m$. Each cell $A[i][j]$ specifies whether test i executed component j. If component j is executed in test i (referred to as component j is *active* in test i), then $A[i][j]$ is 1; otherwise $A[i][j]$ takes the value 0. The error vector E is a n dimensional binary vector. Each cell $E[i]$ corresponds to test-case i and signifies whether the outcome of the test was "pass" or "fail". Usually, but not necessarily, executing a faulty component in a test-case, causes the test to fail. If some test i "fails", then $E[i]$ is set to 1, otherwise $E[i]$ is 0.

Figure 1a shows an example of running SBFL on a program with four components and five tests: the activity matrix A captures when each of the four components is executed in each of the test-cases; e.g., test t_1 executes components C_1, C_2, C_4 and the test outcome was failure (as $E[1]$ is 1).

Once a spectrum is generated, we make use of statistical SBFL metrics (such as Ochiai, Tarantula etc.) to localize the faulty components. The SBFL metrics attempt to analyze the similarity between the activity pattern of a component and the error vector, and assign a *suspiciousness* score to each of the components; a higher score signifies a higher likelihood for a component to be faulty. The components are then ranked based on their suspiciousness scores and the ranked list is used by developers for debugging. Figure 1a ranks components by a popular SBFL metric called Ochiai.

Many such SBFL metrics have been proposed and studied [Wong *et al.*, 2016]. For this work, we select six of the most popular and best studied metrics [Pearson *et al.*, 2017]. Table 1 lists the definitions of these metrics for a component c in the spectrum, where $Pass(c)$ denotes the number of passing test-cases that execute c, $Fail(c)$ signifies the number of failing test-cases that execute c. P and F denote the total number of passing and failing test-cases respectively. In Table 1, note that the $DStar$ metric is built upon the $Kulczynski$ metric by raising the power of the numerator

Table 1: Popular SBFL metrics.

Ochiai	$S(c) = \frac{Fail(c)}{\sqrt{F \cdot (Fail(c)+Pass(c))}}$
Tarantula	$S(c) = \frac{Fail(c)/F}{(Fail(c)/F)+(Pass(c)/P)}$
Op2	$S(c) = Fail(c) - \frac{Pass(c)}{P+1}$
Barinel	$S(c) = 1 - \frac{Pass(c)}{Pass(c)+Fail(c)}$
Kulczynski	$S(c) = \frac{Fail(c)}{Pass(c)+F-Fail(c)}$
DStar	$S(c) = \frac{Fail(c)^*}{Pass(c)+F-Fail(c)}$

A	c1	c2	c3	c4	E
t1	1	1	0	1	1
t2	0	1	0	1	0
t3	1	1	0	0	1
t4	0	1	1	1	1
t5	0	1	0	0	0

Ranked List

	Ochiai
c1	**0.82**
c2	0.77
c4	0.67
c3	**0.58**

(a) Localizing c1 is easy but c3 is difficult, as failing test cases of c1 dominate that of c3.

A	c1	c2	c3	c4	E
t1	1	1	0	1	0
t2	0	1	0	1	0
t3	1	1	0	0	0
t4	0	1	1	1	1
t5	0	1	0	0	0

Ranked List

	Ochiai
c3	**1**
c4	0.58
c2	0.45
c1	0

(b) After fixing c1, localizing c3 becomes easy.

Figure 1: Localizing multiple faulty components in a spectrum.

to "*". The authors suggest a value of $* = 2$ which we use in this work.

For the purpose of debugging, a developer examines the ranked list of components in descending order of their suspiciousness scores. Hence, the *developer effort* to localize a faulty component can be quantified as the fraction of the total program components that one has to examine to reach the faulty component in the ranked list. The objective of fault localization is to minimize the developer effort to catch a fault.

3 Overview

Consider the *spectrum* shown in Figure 1a: we assume that the components C_1 and C_3 are faulty. Consequently, we assume that the tests $\{t_1, t_3, t_4\}$ fail (indicated by the respective error bits in E as 1).

Now, let us analyze this spectrum using Ochiai, one of the most popular SBFL metrics. C_1 emerges as topmost in the ranked list with the highest Ochiai score. Hence, a developer need to check only a single component to localize C_1 as the faulty component with a developer effort of 0.25 (fraction of total components examined to catch C_1). However, identify-

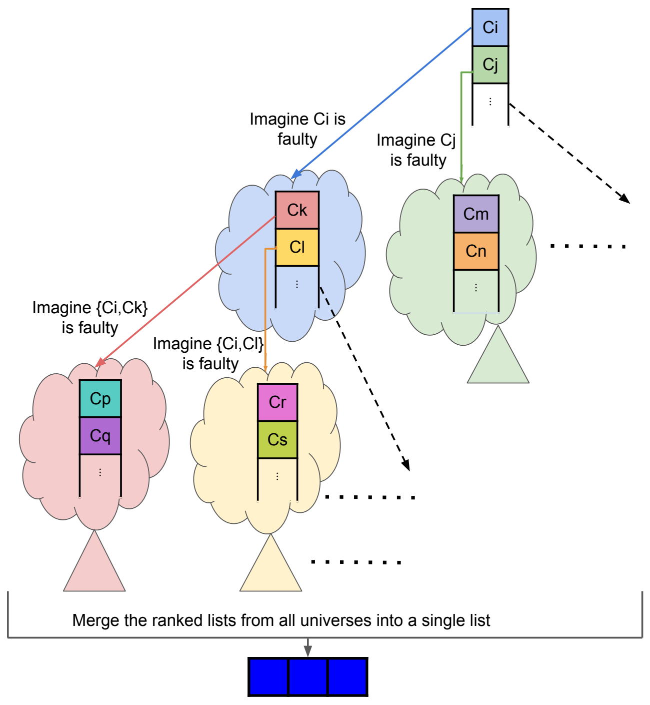
Ci
Cj
Imagine Ci is faulty
Imagine Cj is faulty
Ck
Cl
Cm
Cn
Imagine {Ci,Ck} is faulty
Imagine {Ci,Cl} is faulty
Cp
Cq
Cr
Cs
Merge the ranked lists from all universes into a single list

Figure 2: Our Multiverse Analysis for SBFL.

ing C_3 is more difficult as it has the lowest score, requiring a developer effort of 1.0 (all components need to be examined to arrive at C_3).

C_3 has a low Ochiai score as C_1 *dominates* C_3 on the number of failing tests where they were executed. As Ochiai (and other SBFL metrics) tend to assign higher scores to components that are active in a larger number of failing tests, C_3 ends up lower in the relative ranking.

One way to expose the dormant component C_3 is to adopt a *build-test-rank-fix* cycle. Assume that a developer fixes C_1. Next, she rebuilds the codebase, reruns the test-suite and generates a new spectrum (see Figure 1b). Since, C_1 is fixed, test cases t_1 and t_3 pass and t_4 is the only remaining failing test case as it executed the faulty component C_3 (which is not yet fixed). Ochiai can now rank C_3 at the top of the ranked list; the developer can, now, identify C_3 as a faulty component and fix it. The total developer effort in this case is 0.5.

However, there is one major drawback to this. Every time a buggy component is identified and fixed, the codebase has to be rebuilt and the spectrum has to be regenerated by running the entire test-suite, which is costly. If there are k bugs in a program, the *build-test-rank-fix* cycle has to be repeated k times. This could have been avoided, if we could simulate the process of fixing a bug and regenerating a spectrum.

We design a multiverse analysis approach that *simulates* the *build-test-rank-fix* cycle to localize multiple faults (Figure 2). We start with the initial ranked list generated from the original spectrum of a program. Assume that, C_i has the highest suspiciousness score in the initial list. We, imagine an universe U_i where C_i is the dominant fault. Now in U_i, we try to simulate a repair of C_i. Note that, in Figure 1b, we simulated a repair of C_1 by passing all tests which executed C_1. It was possible as we knew there was no intersection between the execution patterns of the dominant fault C_1 and the dormant fault C_3. In real life, this may not always hold. By rerunning the tests, we would have known exactly which tests fail, but this is not possible in a simulation. Hence, we assume that whenever we simulate a repair of some component C_i, all tests which executed C_i, may still fail with probability p due to some other fault. The problem still remains, however, as computing p is difficult. The activity matrix is usually sparse, the fraction of tests compared to the number of components is low and there are very few failing tests. Further, SBFL is a "black-box" technique and hence we have no information available on the programs being tested other than the spectrum. Hence, we treat p as a hyperparameter and attempt to make a good estimate over a set of training examples with known ground truths. We choose to treat p as a hyperparameter and tune it over a set of benchmarks rather than over individual programs as doing so will result in loss of generality over unknown programs.

Once a repair of C_i is simulated in U_i, we get a new error vector and generate a new ranked list where C_k is the topmost faulty component (Figure 2). However, in real-life scenarios, the topmost ranked component may not necessarily be faulty. So, we repeat the above with the topmost μ components (μ is an user-provided hyperparameter) from the root list assuming that the dominant fault will occur within rank μ. In Figure 2, we also create an universe for the component C_j that is ranked second in the initial ranked list. We continue spawning hypothetical universes until rank μ. For each of the above μ universes, we recursively repeat the process: for example, from the universe U_i, we spawn universes $U_{\{i,k\}}$ where we imagine components $\{C_i, C_k\}$ to be faulty and $U_{\{i,l\}}$ where we imagine components $\{C_i, C_l\}$ to be faulty and so on.

Interestingly, in the universe U_i (where C_i was assumed faulty and fixed), C_k *bubbles up* as the most suspicious component in the new ranked list. Intuitively, conditioning on the assumption that "C_i *is the first fault*" exposes C_k as the second likely faulty component. Further, in another universe, where we imagined both C_i and C_k to be faulty and thus simulated fixing these components, another dormant fault C_p was exposed. We continue this process of spawning such universes until no more failing tests remain in the spectrum.

Our multiverse analysis strategy essentially constructs a *multiverse tree*. The original universe with no assumptions (ranked list generated from the original spectrum) is the root of the tree.

More formally, the initial universe with no assumptions (denoted as $U_\top$) is the standard setting for SBFL. The SBFL score $\mathcal{S}$ of a component C_a is supposed to estimate the probability that C_a is faulty (the SBFL score is proportional to some monotonically increasing function f of the probability):

$$\mathcal{S}^{U_\top}(C_a) \propto f(Prob(C_a \text{ is faulty}))$$

We "lift" these metrics to universes that are conditioned on the assumptions that certain components are faulty:

$$\mathcal{S}^{U_{1,2,\ldots,n}}(C_a) \propto f(Prob(C_a \text{ is faulty} \mid C_1, \ldots, C_n \text{ are faulty}))$$

Once all possible universes have been spawned, we merge the ranked lists across all the universes (the multiverse) into a

Algorithm 1: EXPLORER(A, E, n, S, μ, β, p)

```
Input: Activity Matrix $A$, error Vector $E$, number of components $n$, SBFL metric $S$, number of universes $\mu$, bound on total number of universes $\beta$ and probability of a test failing after a simulated fix $p$
Output: Ranked list of components $\mathcal{L}$
1  Queue $\mathcal{M} \leftarrow \emptyset$, List $\mathcal{L}' \leftarrow \emptyset$, $k \leftarrow 0$
2  $\mathcal{R}_\top \leftarrow$ RANKEDLIST($A$, $E$, $S$)
3  $\mathcal{M}$.enqueue($\langle \mathcal{R}_\top, E \rangle$)
4  $\mathcal{L}'[k] \leftarrow \mathcal{R}_\top$
5  $k$++
6  while $\mathcal{M} \neq \emptyset$ and $k \leq \beta$ do
7      $\langle \mathcal{R}', E' \rangle \leftarrow \mathcal{M}$.dequeue()
8      for $i \leftarrow 1$ to $\mu$ do
9          $\langle \mathcal{R}'', E'' \rangle \leftarrow$ SIMULATE($A$, $E'$, $\mathcal{R}'[i]$, $S$, $p$)
10         if $\mathcal{R}'' \neq \emptyset$ then
11             $\mathcal{M}$.enqueue($\langle \mathcal{R}'', E'' \rangle$)
12             $\mathcal{L}'[k] \leftarrow \mathcal{R}''$
13             $k$++
14 $\mathcal{L} \leftarrow$ MERGE($\mathcal{L}'$, $n$)
15 return $\mathcal{L}$
```

single new ranked list where the dormant faulty components have been ”bubbled up”. While merging, we ensure that the components are ranked according to the highest score that each of them has received over the entire multiverse.

As the multiverse tree can grow exponentially, we use a hyperparameter β as a bound: we stop exploring new universes when either the total number of universes exceed β or there are no more universes to explore.

Application The proposed fault localization strategy can reduce the number of *build-test-rank-fix* iterations by ranking most of the faulty components at the top of the list. Hence, now the developer can examine components up to the top-k ranks in the ranked list produced by the fault localizer, and thereby, fix *many* of the faults in each iteration, thereby reducing the number of *build-test-rank-fix* iterations.

4 Algorithms

We demonstrate our multiverse analysis approach in Algorithm 1. It takes as input a spectrum (activity matrix A and error vector E), number of components in the program n, an underlying SBFL metric S (such as Ochiai, Tarantula etc.). S is a hyperparameter as it is not known which SBFL metric performs the best. Hyperparameter μ denotes that we will only simulate the repair of top-μ components from each ranked list, i.e., each parent universe can have at most μ children universes. Hyperparameter β is a bound on the total number of universes to explore. Hyperparameter p is the probability with which we assume that a test involving a fixed (simulated) component may still fail due to other faults. Algorithm 1 produces a ranked list $\mathcal{L}$ after multiverse analysis.

Our algorithm explores (traverses) the multiverse tree in a breadth-first manner. We begin our exploration by initializing an empty queue $\mathcal{M}$ that will store the universes pending exploration in order of their inception, an empty list $\mathcal{L}'$ that will store all the ranked lists in the multiverse in the order in which

Algorithm 2: SIMULATE(A, E, C, S, p)

```
Input: Activity Matrix $A$, error Vector $E$, component $C$ which is imagined to be faulty, SBFL metric $S$, probability of a test failing after a simulated fix $p$
Output: Tuple of new ranked list and error vector $\langle \mathcal{R}, E \rangle$
1  foreach row $\rho$ in $A$ do
2      if $A[\rho][C] = 1$ and $E[\rho] = 1$ then
3          /*if test $\rho$ executed $C$ and it failed, assume $C$ is fixed and sample the error bit $E[\rho]$ from the outcome of a Bernoulli trial with a probability of success $p$*/
4          $E[\rho] \leftarrow Bernoulli(p)$
5  if $E = \vec{0}$ then
6      /*if all test outcomes are “success”, return “spurious” universe*/
7      return $\langle \emptyset, \vec{0} \rangle$
8  else
9      $\mathcal{R} \leftarrow$ RANKEDLIST($A$, $E$, $S$)
10     return $\langle \mathcal{R}, E \rangle$
```

they were generated and a counter k which keeps track of how many universes have been explored so far (Line 1). Next, we generate the first ranked list $\mathcal{R}_\top$ where no component is assumed to be faulty, by invoking the procedure *RankedList* with arguments A, E and S (Line 2). The *RankedList* procedure returns a ranked list of components and their suspiciousness scores computed using S (see Figure 1a). A universe is a tuple of a ranked list and an error vector; we enqueue the first universe $\langle \mathcal{R}_\top, E \rangle$ in the multiverse queue $\mathcal{M}$ (Line 3). We also add the first ranked list $\mathcal{R}_\top$ in $\mathcal{L}'$ (Line 4). Next, we explore all the unvisited universes in queue $\mathcal{M}$ until either $\mathcal{M}$ is empty, or we exceed the bound β.

To explore a universe we dequeue one universe ($\langle \mathcal{R}', E' \rangle$) from $\mathcal{M}$ (Line 7). Next, for each component in $\mathcal{R}'$ upto rank μ, we invoke the *Simulate* procedure in Line 9. The *Simulate* procedure takes as input A, E', $R'[i]$, the i^{th} component which will be assumed to be faulty, S, p and returns a new universe $\langle \mathcal{R}'', E'' \rangle$ where a repair of $R'[i]$ has been simulated. If $\mathcal{R}''$ is empty, then the new universe is spurious, i.e., it contains no failing test cases and we do not consider it for further simulation. Otherwise, we enqueue the new universe in $\mathcal{M}$ (Line 11), add the new ranked list $\mathcal{R}''$ to $\mathcal{L}'[k]$ (Line 12) and continue the process.

Once we have explored the multiverse, all the ranked lists in $\mathcal{L}'$ are merged into a single ranked list by invoking *Merge* (Line 14) and returned.

The *Simulate* procedure is shown in Algorithm 2. It takes as input A, E, S, a component C which is to be assumed faulty and p. Algorithm 2 simulates a repair of C and returns a new ranked list R where faulty components dominated by C should bubble up; along with a new simulated error vector E. Algorithm 2 starts by iterating over each row (test case) in A. For each test ρ, if ρ executes C, i.e., $A[\rho][C]$ is 1 and if it failed, i.e., $E[\rho]$ is 1, then the new outcome of ρ, assuming C is fixed, is decided by a *Bernoulli* trial with a probability of success p (Line 4). If the outcome of the *Bernoulli* trial is failure, i.e., 0, then we assume that the test ρ passes after a simulated repair of C, i.e., $E[\rho]$ is also 0. Otherwise, if the

Algorithm 3: MERGE($\mathcal{L}'$, n)

```
Input: List of all ranked lists in multiverse $\mathcal{L}'$, number of
       components $n$
Output: Ranked list of components $\mathcal{L}$
1 List $\mathcal{M} \leftarrow \vec{0}$
2 for $i \leftarrow 1$ to $\mathcal{L}'.size$ do
3     for $j \leftarrow 1$ to $n$ do
4         $c \leftarrow \mathcal{L}'$[i][j]        //component of rank $j$ in list $i$
5         $score \leftarrow \mathcal{L}'$[i][j].score
6         if $score > \mathcal{M}[c]$ then
7             $\mathcal{M}[c] \leftarrow score$
8 return SORTANDRANK($\mathcal{M}$)
```

Table 2: Details of the spectrums used in the experiments.

Project	Spectrums	% of spectrums with k faults			
		$k = 1$	$k = 2$	$k = 3$	$k \geq 4$
Chart	50	26%	24%	16%	34%
Closure	68	47%	32%	9%	12%
Lang	139	30%	43%	19%	8%
Math	378	38%	34%	14%	14%
Mockito	48	25%	58%	17%	0%
Time	89	62%	21%	10%	7%

outcome of the trial is success, i.e., 1, we assume that test ρ may still fail due to other faults and set $E[\rho]$ to 1. Next, we check whether the new error vector E contains any failing tests. If there is none, then we return a spurious universe $\langle \emptyset, \vec{0} \rangle$ (Line 7). Otherwise, we generate a new ranked list R by invoking *RankedList* with the modified error vector E, A and S (Line 9). Finally we return this new universe as a tuple $\langle \mathcal{R}, E \rangle$ (Line 10).

Algorithm 3 demonstrates the *Merge* procedure which takes as input $\mathcal{L}'$, the list of all ranked lists in the multiverse and number of components n (number of columns in A) and produces a single ranked list $\mathcal{L}$. It starts by initiating a list $\mathcal{M}$ with $\vec{0}$ (Line 1) which is used to store the highest score of each component over the entire multiverse.

Next, we iterate over each list $\mathcal{L}'[i]$ from the multiverse (Line 2). For each component c at rank j (Line 3) in $\mathcal{L}'[i]$, we check whether the score of c in $\mathcal{L}'[i]$ is greater than its highest score $\mathcal{M}[c]$ that have been seen so far (Line 6). If true, then we update $\mathcal{M}[c]$ with the new high score, otherwise we move on to the next component without an update. At the end, $\mathcal{M}[i]$ contains the highest score of component i over the entire mutiverse. Finally, we sort the suspiciousness scores in $\mathcal{M}$ and return the final ranked list of components (Line 8).

Note that, there can be many possible ways to merge the ranked lists in the multiverse. Algorithm 3 is one among such possibilities which we have found to be effective.

5 Empirical Analysis

We implement the proposed multiverse analysis approach in a tool called ARTEMIS. To evaluate the effectiveness of our proposed approach, we experimentally evaluate the following research questions:

- **RQ1:** What are the optimal choices of hyperparameters for ARTEMIS?
- **RQ2:** How does ARTEMIS compare against the most popular SBFL metrics?

Experimental Subjects. We have performed our experiments on six java project repositories, *Chart*, *Closure*, *Lang*, *Math*, *Mockito* and *Time* from the DEFECTS4J benchmark suite [Just *et al.*, 2014]. These are large open source softwares containing $22,000$ to $96,000$ lines of code and have been developed over a decade. Overall, these repositories contain 395 real life software bugs and each bug correspond to one or more faults. The ground truth for each bug is available for fault localization studies. DEFECTS4J has been exclusively used in many recent fault localization studies [Pearson *et al.*, 2017; Chatterjee *et al.*, 2020]. There exists other buggy datasets such as (SIR) [Do *et al.*, 2005] or Siemens suite [Hutchins *et al.*, 1994], however, we have not used these as the size of the programs are very small (around only 100 to 500 lines of code) and the bugs do not represent real world behavior as they are either hand-crafted or simple mutations.

Spectrums. We have first collected the spectrums generated in [Chatterjee *et al.*, 2020] for the DEFECTS4J dataset v1.4.0[1]. The spectrums were generated by EVOSUITE [Fraser and Arcuri, 2011]. To ensure the diagnostic quality of test-suites, the state-of-the-art fitness functions ULYSIS [Chatterjee *et al.*, 2020], DDU [Perez *et al.*, 2017] and COVERAGE were used as the search criteria for the evolutionary algorithm in EVOSUITE. The spectrums were generated at line granularity, i.e., each component (column) in an activity matrix correspond to a line of code. Spectrums without failing tests were discarded as fault localization cannot be performed on such instances. Overall, we use 772 spectrums. Additional details are provided in Table 2.

Note that, we elect to use an automated test-generation framework as our objective is to utilize ARTEMIS to address the shortcomings of an automated fault localization pipeline. Automated testing has been found to be very effective in providing more coverage and exposing unexpected faults in programs [Serra *et al.*, 2019]. Manually written test suites are good at exposing faults but they also incur additional cost for employing developers.

5.1 Evaluation Metrics

$\mathcal{EXAM}$ score is one of the popular choices to compare fault localization performance. $\mathcal{EXAM}$ score is defined as k/m, where k is the rank of the faulty component and m is the total number of components in the program; a low $\mathcal{EXAM}$ score indicates an effective fault localization technique. Note that, there exist many other similar metrics which quantify developer effort for debugging. We choose to use $\mathcal{EXAM}$ score as it provides a human-interpretable score in terms of components checked by a developer to catch a fault [Wong *et al.*, 2016].

We also used the top-n [Li and Zhang, 2017; Pearson *et al.*, 2017], also known as acc@n [B. Le *et al.*, 2016] or

[1]`https://github.com/rjust/defects4j/tree/v1.4.0`

Table 3: P-values for the hypothesis test between ARTEMIS and the base SBFL metrics on 1526 faults from 772 spectrums over all projects.

Ochiai	D^2	Barinel
$1.75e^{-11}$	$7.35e^{-14}$	$9.24e^{-13}$
Op2	**Tarantula**	**Kulczynski**
$5.27e^{-21}$	$8.14e^{-12}$	$4.59e^{-12}$

Einspect@n [Zou *et al.*, 2021], which counts the number or percentage of faults successfully localized within the top-n ranks.

5.2 RQ1: Hyperparameter Selection

Our multiverse analysis (Algorithm 1) uses the following hyperparameters: μ, β, the underlying SBFL metric S and p. To select the best hyperparameter combination and to show that the configuration generalizes, we use k-fold cross validation. We construct six folds over the set of spectrums for each of the six projects. In each fold, we set aside the spectrums from one project as test data and search for the best hyperparameters over the spectrums from the remaining five projects. This ensures that the performance of ARTEMIS generalizes when being tested on unknown projects.

We employ a grid search to select the best value of hyperparameters. We search for μ in a range from 1 to 20. Since, μ is the maximum number of children for each node in the multiverse tree, it has an exponential effect on the size of the tree and going any higher significantly slows down ARTEMIS. For β, we restrict the search in $\{5, 10, 15, 20, \ldots, 100\}$. A higher value of β has no noticeable impact on the performance of ARTEMIS as on real softwares, the number of failing tests are often very small and all failing tests are usually exhausted well before a value of 100. We search for the best SBFL metric S to use in ARTEMIS from the set $\{Ochiai$, D^2, $Barinel$, $Op2$, $Tarantula$, $Kulczynski\}$ as these had been identified as the top performing SBFL metrics in a recent study [Pearson *et al.*, 2017]. Finally, for p, the probability with which a test may fail after a simulated repair, we search for an optimal value within $\{1e-5, 1e-4, \ldots, 1e-1\}$. We constrain the search for p within a set of small values as an execution path containing multiple faults in a well tested software repository, i.e. where p can be greater than 0, is rare. Hence, a high value for p may not generalize well. For each of the six folds, we select the best hyperparameter combination for ARTEMIS which provides the lowest $\mathcal{EXAM}$ score on average on the training data and use it to localize faults from the spectrums of the test data, i.e., the spectrums from the unknown project. This is a standard problem in any statistical/machine learning task on adapting hyperparameters to new datasets. Generally, we assume that some ground truths are available on the new datasets for tuning the hyperparameters. Finally, we compare the performance of ARTEMIS against each of the base SBFL metrics on the test data.

For each fold, we select the hyperparameter combinations that provide the lowest $\mathcal{EXAM}$ score on average on the training data and rank the faulty components on the test data. ARTEMIS provides an average improvement of 21%, 11%, 17%, 11%, 23%, 10% respectively on test data *Chart*, *Closure*, *Lang*, *Math*, *Mockito*, *Time* over the base SBFL metrics. ARTEMIS performs better than each base SBFL metric in each fold which signifies that ARTEMIS generalizes well on unknown buggy spectrums. The most prevalent hyperparameter combinations over all of the six folds are $\{S =$ Ochiai, $\mu = 17$, $\beta = 20$, $p = 1e-5\}$ (detailed results in appendix).

5.3 RQ2: Comparison against Most Popular SBFL Metrics

To determine how ARTEMIS fares against the base SBFL metrics we compare the $\mathcal{EXAM}$ score for each faults using ARTEMIS (with the most prevalent hyperparameter combination; $S =$ Ochiai, $\mu = 17$, $\beta = 20$, $p = 1e-5$) against each of the base SBFL metrics.

First, we conduct an one-tailed paired Wilcoxon Signed-rank test [Rosner *et al.*, 2006] between the $\mathcal{EXAM}$ score for each fault by ARTEMIS against that of each base SBFL metric. To perform the statistical tests, we denote the vector of $\mathcal{EXAM}$ scores using ARTEMIS as X and for each base metric i, we denote the same as Y_i. Next, we test the null hypothesis $\mathcal{H}_0$: the median of $(X - Y_i)$=0 against the alternative hypothesis $\mathcal{H}_1$: the median of $(X - Y_i) < 0$. If we can refute $\mathcal{H}_0$ with 99% confidence interval, we infer that ARTEMIS is statistically better than the base metric i (similar strategy is followed in [Chatterjee *et al.*, 2020]). We do not conduct a statistical test for each project separately as the number of spectrums in some projects are too small for a reliable statistical test. Instead we perform the hypothesis testing over 1526 faulty components contained in 772 spectrums. We found that the performance of ARTEMIS is statistically better than every base SBFL metric (see Table 3).

Table 4 presents the mean (with standard deviation) and median percentage improvement in developer effort (in terms of $\mathcal{EXAM}$ score) that ARTEMIS provides over each of the base SBFL metrics. Overall, ARTEMIS provides approximately 17% mean and 14% median improvement over the base SBFL metrics on average.

Table 5 demonstrates the percentage of faulty components that are ranked within top-n positions in the ranked lists using any approach. Note that , ARTEMIS is as good as or slightly better than the base SBFL metrics while placing dominant faults within the first rank. ARTEMIS places 8% of the faults at the first rank which is also achieved by Ochiai and D^2 while the rest perform slightly worse. This indicates that ARTEMIS does not push down dominant faults in an effort to bubble up the dormant faults. The effect of bubbling up dormant faults can be seen from $n = 2$ and onward. ARTEMIS places 17% of the faults within rank 2 or less, whereas Ochiai, D^2, Barinel and Tarantula could place only 12%. Similarly, ARTEMIS places 31% of the faults within rank 5, whereas Ochiai places only 26% and the rest performs even worse. This demonstrates the effectiveness of our proposed approach (project-wise comparison for Tables 4 and 5 provided in appendix).

6 Related Work

One-fault-at-a-time approaches work by examining the ranked list until a developer catches a fault. Next, this fault is

Table 4: Mean (with standard deviation) and median percentage improvement in developer effort ($\mathcal{EXAM}$ score) using ARTEMIS over the base SBFL metrics on 1526 faults from 772 spectrums from all projects.

	Ochiai	D^2	Barinel	Op2	Tarantula	Kulczynski
Mean(s.d)	16(3.2)%	24(4.7)%	17(2.9)%	27(5.1)%	17(2.9)%	18(4.5)%
Median	13%	19%	14%	22%	14%	14%

Table 5: Percentage of faulty components that are ranked within top-n. Number of faults over 772 spectrums is 1526.

top-n	ARTEMIS	Ochiai	D^2	Barinel	Op2	Tarantula	Kulczynski
$n = 1$	**8%**	**8%**	**8%**	6%	5%	6%	7%
$n = 2$	**17%**	12%	12%	12%	10%	12%	11%
$n = 5$	**31%**	26%	25%	24%	20%	24%	25%
$n = 10$	**50%**	46%	42%	41%	37%	41%	46%
$n = 20$	**69%**	65%	61%	63%	54%	63%	65%
$n = 50$	**87%**	83%	80%	86%	76%	86%	84%

repaired, the test-suite is rerun and the hunt for next fault begins anew by re-ranking the components. This approach has seen application in the DStar technique [Wong *et al.*, 2013] and Bayesian reasoning techniques [Abreu *et al.*, 2009c]. One disadvantage associated with such approaches is the cost of rerunning the test-suite every time a fault is fixed. We sidestep this problem entirely with effective simulation of the "fix-and-re-rank" strategy. Furthermore, one potential weakness of most techniques using Bayesian reasoning is the underlying assumption that multiple faulty components fail independently which may not always hold in real-life scenarios [Wong *et al.*, 2016].

Multiple fault localization techniques usually attempt to rank entire sets of components based on their collective suspiciousness rather than ranking single components, e.g., [Abreu *et al.*, 2009b; Abreu *et al.*, 2011] propose an approach that generates a ranked list of multi-candidate diagnoses.

Multi-fault localization techniques which aim to produce diagnostic reports on multi-component candidates are orthogonal to our strategy, i.e., all these techniques attempt to rank multi-component faulty candidates. Furthermore, these techniques (like [Abreu *et al.*, 2009b; Abreu *et al.*, 2011]) require expensive hitting set computations and therefore do not scale well and have only been demonstrated on small programs from the SIEMENS suite.

[Jeffrey *et al.*, 2009] proposed a fault localization method based on value replacement for reducing the time to localize multiple bugs by using an iterative approach to rank and localize multiple bugs. However, the method has a high computational overhead, which can be costly in larger subject programs. [Wei and Han, 2013] proposed a parameter combination approach coined PBC to aid in localizing bugs in multiple bug programs. The authors used a bisection method to cluster failed test cases with crosstab-based fault localization technique. The result shows that PBC performs better than Tarantula [Jones *et al.*, 2002]. [Steimann and Frenkel, 2012] used partitioning procedures from integer linear programming to improve SBFL efficacy on multi-fault programs. This method breaks down the fault localization problem into various partitions to facilitate localization by multiple developers. One issue with clustering based approaches is that they assume a precise causal relationship between faults and their execution profiles, i.e., test-cases which trigger the same buggy component will exhibit similar activity patterns. However, this assumption may not always hold [Wong *et al.*, 2016].

There exist earlier studies that have used rank refinement strategies. A tester feedback driven approach [Bandyopadhyay and Ghosh, 2012] was proposed to gradually filter false positives (tests which execute faulty components, but do not trigger failure) from the test-suite in order to refine rankings. This proved to be comparable to Ochiai on the Siemens and Unix utilities suite. A neural network based approach was proposed [Zhao *et al.*, 2022] for boosting SBFL metric rankings on Defects4J benchmarks by performing analysis on method interactions in addition to the spectrums. Another approach used a combination of SBFL metrics [Majd *et al.*, 2022] in order to boost the overall ranking of faulty components in the Siemens benchmark suite.

Multiverse analysis has been previously used for generating "good" test-suites [Chatterjee *et al.*, 2020] which is orthogonal to the problem of fault localization. Chatterjee et al. used multiverse analysis to quantify the diagnostic quality of a test-suite to lead an evolutionary search for a "good" test-suite. For a program with m components, this approach explores exactly m universes with the inherent assumption that the program contains a single fault. In contrast, we explore a multiverse tree in order to bubble up dormant faults in a multi-fault scenario.

7 Conclusion

We propose an approach that simulates repair of dominating faulty components so that the dormant faults in a program can be exposed earlier. We implement our ideas into a tool named ARTEMIS, and experiment with six of the most popular and widely studied SBFL metrics as the baseline. We show that the hyperparameters of ARTEMIS generalize well over unknown data. Further, ARTEMIS provides a statistically significant improvement of approximately 17% mean and 14% median over the base SBFL metrics in terms of developer effort ($\mathcal{EXAM}$ score) on average.

Threats to validity We performed experiments on six real-life softwares and generated tests using Coverage, DDU and ULYSIS which are three most popular fitness functions for evolutionary search based test generation using EVOSUITE. Though we believe that our study generalizes beyond the environment in which it was performed, experiments to replicate this study in other environments can be conducted. Further, we used the most popular SBFL metrics as the baseline, but experiments on other heuristics can also be performed.

Acknowledgments

We are thankful to Intel for their support via the Intel India Research Fellowship Program and the guidance of the mentors from the organization. We also wish to thank Google Research for their support. This work was also partially supported by FCT through the LASIGE Research Unit (ref. UIDB/00408/2020 and ref. UIDP/00408/2020) and INESC-ID (ref. UIDB/50021/2020).

References

[Abreu *et al.*, 2009a] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. A practical evaluation of spectrum-based fault localization. *Journal of Systems and Software*, 82(11):1780–1792, 2009.

[Abreu *et al.*, 2009b] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. Localizing software faults simultaneously. In *2009 Ninth International Conference on Quality Software*, pages 367–376. IEEE, 2009.

[Abreu *et al.*, 2009c] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. Spectrum-Based Multiple Fault Localization. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 88–99, 2009.

[Abreu *et al.*, 2011] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. Simultaneous debugging of software faults. *Journal of Systems and Software*, 84(4):573–586, 2011.

[B. Le *et al.*, 2016] Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. A Learning-to-Rank Based Fault Localization Approach Using Likely Invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 177–188, New York, NY, USA, 2016. Association for Computing Machinery.

[Bandyopadhyay and Ghosh, 2012] Aritra Bandyopadhyay and Sudipto Ghosh. Tester feedback driven fault localization. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 41–50, 2012.

[Chatterjee *et al.*, 2020] Prantik Chatterjee, Abhijit Chatterjee, José Campos, Rui Abreu, and Subhajit Roy. Diagnosing software faults using multiverse analysis. In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, pages 1629–1635. International Joint Conferences on Artificial Intelligence Organization, 7 2020. Main track.

[Choi *et al.*, 2010] Seung-Seok Choi, Sung-Hyuk Cha, and Charles C Tappert. A survey of binary similarity and distance measures. *Journal of systemics, cybernetics and informatics*, 8(1):43–48, 2010.

[Do *et al.*, 2005] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[Fraser and Arcuri, 2011] Gordon Fraser and Andrea Arcuri. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software. In *Proc. of the 19th ACM ESEC/FSE*, page 416–419. ACM, 2011.

[Hutchins *et al.*, 1994] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments on the effectiveness of dataflow-and control-flow-based test adequacy criteria. In *Proceedings of 16th International conference on Software engineering*, pages 191–200. IEEE, 1994.

[Jeffrey *et al.*, 2009] Dennis Jeffrey, Neelam Gupta, and Rajiv Gupta. Effective and efficient localization of multiple faults using value replacement. In *2009 IEEE International Conference on Software Maintenance*, pages 221–230. IEEE, 2009.

[Jones and Harrold, 2005] James A Jones and Mary Jean Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282, 2005.

[Jones *et al.*, 2002] James A Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering. ICSE 2002*, pages 467–477. IEEE, 2002.

[Just *et al.*, 2014] René Just, Darioush Jalali, and Michael D. Ernst. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *Proc. of the 23rd ISSTA*, page 437–440. ACM, 2014.

[Li and Zhang, 2017] Xia Li and Lingming Zhang. Transforming Programs and Tests in Tandem for Fault Localization. *Proc. ACM Program. Lang.*, 1(OOPSLA), oct 2017.

[Majd *et al.*, 2022] Amirabbas Majd, Mojtaba Vahidi-Asl, Alireza Khalilian, and Babak Bagheri. Consilientsfl: using preferential voting system to generate combinatorial ranking metrics for spectrum-based fault localization. *Applied Intelligence*, pages 1–21, 2022.

[Naish *et al.*, 2011] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A model for spectra-based software diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)*, 20(3):1–32, 2011.

[Pearson *et al.*, 2017] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. Evaluating and improving fault localization. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 609–620. IEEE, 2017.

[Perez *et al.*, 2017] Alexandre Perez, Rui Abreu, and Arie van Deursen. A test-suite diagnosability metric for spectrum-based fault localization approaches. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 654–664. IEEE, 2017.

[Rosner *et al.*, 2006] Bernard Rosner, Robert J Glynn, and Mei-Ling T Lee. The wilcoxon signed rank test for paired comparisons of clustered data. *Biometrics*, 62(1):185–192, 2006.

[Serra *et al.*, 2019] Domenico Serra, Giovanni Grano, Fabio Palomba, Filomena Ferrucci, Harald C Gall, and Alberto Bacchelli. On the effectiveness of manual and automatic unit test generation: ten years later. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 121–125. IEEE, 2019.

[Steimann and Frenkel, 2012] Friedrich Steimann and Marcus Frenkel. Improving coverage-based localization of multiple faults using algorithms from integer linear programming. In *2012 IEEE 23rd International Symposium on Software Reliability Engineering*, pages 121–130. IEEE, 2012.

[Wei and Han, 2013] Zheng Wei and Bai Han. Multiple-bug oriented fault localization: A parameter-based combination approach. In *2013 IEEE Seventh International Conference on Software Security and Reliability Companion*, pages 125–130. IEEE, 2013.

[Wong *et al.*, 2013] W Eric Wong, Vidroha Debroy, Ruizhi Gao, and Yihao Li. The dstar method for effective software fault localization. *IEEE Transactions on Reliability*, 63(1):290–308, 2013.

[Wong *et al.*, 2016] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, 2016.

[Zhao *et al.*, 2022] Guyu Zhao, Hongdou He, and Yifang Huang. Fault centrality: boosting spectrum-based fault localization via local influence calculation. *Applied Intelligence*, 52(7):7113–7135, 2022.

[Zou *et al.*, 2021] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D. Ernst, and Lu Zhang. An Empirical Study of Fault Localization Families and Their Combinations. *IEEE Transactions on Software Engineering*, 47(2):332–347, 2021.