Prevalence of Single-Fault Fixes and its Impact on Fault Localization

Alexandre Perez*†, Rui Abreu*†, Marcelo d'Amorim‡
*University of Porto, Portugal
†Palo Alto Research Center, USA
‡Federal University of Pernambuco, Brazil
alexandre.perez@fe.up.pt, rui@computer.org, damorim@cin.ufpe.br

***Abstract*—Several fault predictors were proposed in the context of Spectrum-based Fault Localization approaches to rank software components in order of suspiciousness of being the root-cause of observed failures. Previous work has also shown that some of the fault predictors (near-)optimally rank software components, provided that there is one fault in the system. Despite this, further work is being spent on creating more complex, computationally expensive, model-based techniques that can handle multiple-faulted scenarios accurately. However, our hypothesis is that when software is being developed, bugs arise one-at-a-time and therefore can be considered as single-faulted scenarios. We describe an approach to mine repositories, find bug-fixes, and catalog them according to the number of faults they fix, to assess the prevalence of single-fault fixes. Our empirical study using 279 open-source projects reveals that there is a prevalence of single-fault fixes, with over 82% of all fixes only eliminating one bug from the system, enabling the use of simpler, (near-)optimal, fault predictors. Moreover, we draw on the practical implications of our findings to influence and set direction for future research.**

I. Introduction

Spectrum-based fault localization is a popular approach to efficiently and effectively debug software applications [1]. These techniques use per-test coverage information to correlate component involvement to test outcomes (pass/fail) [2]. Over the years, several authors have contributed to the maturity of these techniques by studying the impact of different scoring mechanisms, commonly referred to as *fault predictors*, in the diagnostic accuracy [1], [3], [4], [5], [6], [7].

The O predictor proposed by Abreu *et al.* [8] and later corroborated by Naish *et al.* [9], has been shown to be the *optimal* heuristic for locating faults provided the system under analysis contained only one fault [9]. However, in the eventuality of the system containing multiple faults, the performance of O is expected to degrade considerably, as it assumes that one component must be responsible for all failing tests [8]. For this reason, previous research has focused on scenarios of multiple faults by either proposing less optimal fault predictors whose performance does not degrade as severely in the presence of multiple faults (*e.g.*, the D^* predictor [10]); by proposing more intricate techniques, that are both computationally expensive and dependent on a behavioral model of the system [8], [11] or by clustering tests so that there are multiple single-faulted subproblems that current fault predictors can handle [12], [13]. All these approaches add complexity to the debugging process.

However, what remains to be seen is how often such multiple-bug scenarios actually happen in practice. Our hypothesis is that more often than not programmers detect and fix *one bug at a time* during development. This would mean that, most often, developers are faced with single-faulted scenarios, so the use of the optimal fault predictor O could be justified given the prevalence of single-faults and thus we would be able to provide optimal diagnostic reports to developers and also make better use of tools and techniques that take diagnostic reports as inputs, such as with program repair [14]. Also note that our hypothesis does not state that the system is single-faulted but rather that faults are mostly detected in isolation.

To assess that such single-fault fix prevalence actually exists in practice, we describe a methodology that mines a project's code repository to find bug fixes and label them as being single- or multiple-faulted. The repository miner performs a reverse-chronological exploration of commits and runs newer test suites against older versions of the program. If a passing test-suite fails against an older, it means that the code changes between the two versions (*i.e.*, Δ) contain a fix. If, on the other hand, there are compilation or runtime errors while running tests (due to, *e.g.*, a change in the interface between components), then we consider Δ as adding new functionality — so there is no fix present. Our classifier will then find if there is any component in Δ that appears in every affected test. If so, the fix is considered to be single-faulted. Otherwise, the fix will be labeled as multiple-faulted. Our methodology is similar to those of Böhme *et al.* [15], Dallmeier *et al.* [16] and Sliwerski *et al.* [17], in that code repositories are explored to isolate fixes.

We conducted a large-scale empirical study where we analyzed the repositories of 279 real, open-sourced Java projects, catalogued every detected fix, and performed fault-localization using 5 popular predictors. In total, 1375 fixes were found. Out of all fixes, 1135 of them were single-faulted, thus yielding a prevalence of 82.5%. Among single-faulted fixes we observed that the O predictor has the best accuracy out of the tested predictors, with the faulted component being placed at the top of the diagnostic report in over 90% of all cases. Additionally, we found that another predictor proposed in the literature [9] (O^P, a non-optimal variant of O) performed similarly to O, while other predictors were less accurate. For multiple-faulted fixes, the diagnostic performance of O decreased

drastically, making its fault localization reports unsuited for analysis. Other predictors showed a less severe performance degradation.

After analyzing the results, we have verified our hypothesis that most failures developers face are due to only one (active) bug, as there is a prevalence of single-fault fixes. However, our results suggest that the optimal O predictor's accuracy deteriorates significantly in the presence of multiple faults. On the upside, the O^P fault predictor has shown comparable performance to the optimal O in the case of single-faults, while still producing usable results for diagnosing multiple-faults.

This paper's contributions are:

- A methodology for finding fixes in a software repository and labeling them as single- or multiple-faulted.
- Empirical evidence that single-faulted fixes correspond to 82.5% of all fixes in open-source Java projects.
- An assessment of the diagnostic performance of spectrum-based fault predictors in single-faulted scenarios. The optional O predictor, as well as O^P, show a degree of accuracy (with virtually no wasted effort) when compared to other predictors.
- An assessment of diagnostic performance in multiple-fault scenarios. We show that O's performance is essentially random. For other predictors, there is still a performance decrease, not as significant as O's, especially when trying to find the last faulty component in the ranking.

To foster reproducibility, the repository miner for classifying fixes is available at https://github.com/aperez/single-fault-prevalence.

II. Background & Related Work

In this section, we establish definitions and basic assumptions held throughout this work. We also outline past research on spectrum-based fault localization. We use the following terminology defined by Avižienis *et al.* [18]:

Definition 1 (Failure)**.** An event that occurs when delivered service deviates from correct service.

Definition 2 (Error)**.** A system state that may cause a failure.

Definition 3 (Fault)**.** The cause of an error in the system.

We apply this terminology to software programs, where faults correspond to defects/bugs in the program code. Failures and errors are symptoms caused by faults in the program. The purpose of fault localization is to pinpoint the root cause of observed symptoms as to guide fixes.

A. Spectrum-based Analysis

Spectrum-based fault localization is an approach to pinpointing bugs in software programs [1], [3], [5], [19]. In spectrum-based fault localization, the following is given: a finite set $\mathcal{C} = \{c_1, c_2, ..., c_M\}$ of M system *components*[1]; a finite set $\mathcal{T} = \{t_1, t_2, ..., t_N\}$ of N system transactions, which correspond to records of a system execution, such as, *e.g.*, test cases; the outcome of system transactions is encoded in the error vector $e = \{e_1, e_2, ..., e_N\}$, where $e_i = 1$ if transaction t_i has failed and $e_i = 0$ otherwise; and a $N \times M$ coverage matrix $\mathcal{A}$, where $\mathcal{A}_{ij}$ encodes the involvement of component c_j in transaction t_i.

[1]A component can be any source code artifact of arbitrary granularity such as a class, a method, a statement, or a branch [2].

The pair $(\mathcal{A}, e)$ is commonly referred to as spectrum [2], and is depicted in Figure 1. Several types of spectra exist. The most commonly used is called hit-spectrum, where the coverage matrix is encoded in terms of binary *hit* (1) and *not hit* (0) flags, i.e., $\mathcal{A}_{ij} = 1$ if c_j is involved in t_i and $\mathcal{A}_{ij} = 0$ otherwise.

$\mathcal{T}$	c_1	c_2	$\cdots$	c_M	e
t_1	$\mathcal{A}_{11}$	$\mathcal{A}_{12}$	$\cdots$	$\mathcal{A}_{1M}$	e_1
t_2	$\mathcal{A}_{21}$	$\mathcal{A}_{22}$	$\cdots$	$\mathcal{A}_{2M}$	e_2
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
t_N	$\mathcal{A}_{N1}$	$\mathcal{A}_{N2}$	$\cdots$	$\mathcal{A}_{NM}$	e_N

Fig. 1: An example spectrum.

The pair $(\mathcal{A}, e)$ serves as input to the fault localization technique. With this input, the next step in this coverage-based technique consists of determining what columns of the matrix A resemble the error vector e the most. For that, an intermediate component frequency aggregator is computed:

$$n_{pq}(j) = |\{i \mid \mathcal{A}_{ij} = p \wedge e_i = q\}| \quad (1)$$

where $n_{pq}(j)$ is the number of runs in which the component j has been active during execution ($p = 1$) or not ($p = 0$), and in which the runs failed ($q = 1$) or passed ($q = 0$). For instance, $n_{11}(j)$ counts the number of times component j has been involved ($p = 1$) in failing executions ($q = 1$), whereas $n_{10}(j)$ counts the number of times component j has been involved in passing executions.

We then calculate similarity to the error vector by means of applying *fault predictors* to each component to produce a score quantifying how likely it is to be faulty. Components are then ranked according to such likelihood scores and reported to the user.

B. Fault Predictors

Table I details fault predictors that are amongst the best performing ones in related work [1]. All of these fault predictors will score a component j so that it informs the fault localization technique on how to produce a ranked list of components for user inspection.

A fault predictor named DStar (D^*) is reported by Wong *et al.* [10] such that the likelihood of a component j being faulty is: (1) proportional to the number of failed tests that cover it (2) inversely proportional to the number of successful tests that cover it; and (3) inversely proportional to the number of failed tests that do not cover it. Wong *et al.*'s intuition is that statement (1) and should carry a higher weight than statements (2) and (3). Therefore, DStar provides a $*$ parameter — where

TABLE I: Fault predictor formulas.

Predictor	Formula
D* [20], [10]	$\frac{n_{11}(j)^*}{n_{01}(j)+n_{10}(j)}$
O [8], [9]	$\begin{cases} -1 & \text{if } n_{01}(j) > 0 \\ n_{00}(j) & \text{otherwise} \end{cases}$
O^P [9]	$n_{11}(j) - \frac{n_{10}(j)}{n_{10}(j)+n_{00}(j)+1}$
Ochiai [4]	$\frac{n_{11}(j)}{\sqrt{n_{11}(j)+n_{01}(j)+\sqrt{n_{11}(j)+n_{10}(j)}}}$
Tarantula [3]	$\frac{\frac{n_{11}(j)}{n_{11}(j)+n_{01}(j)}}{\frac{n_{11}(j)}{n_{11}(j)+n_{01}(j)}+\frac{n_{10}(j)}{n_{10}(j)+n_{00}(j)}}$

$* \geq 1$ — for changing the weight carried by $n_{11}(j)$ in the formula's numerator. In this paper, we use $* = 2$, since [20] claims that D^2 is more effective as a fault predictor than other similarity coefficients.

The O fault predictor is often called the *optimal* metric, but it assumes that there is only one fault affecting the system [9]. Given only one bug, then its $n_{01}(j)$ should always be zero, and therefore any component with a nonzero $n_{01}(j)$ is given the lowest score. Since $n_{11}(j) + n_{01}(j)$ equals the number of failing runs, and $n_{10}(j)+n_{00}(j)$ equals the number of passing runs, there is only one degree of freedom left, expressed by assigning $n_{00}(j)$ as the predictor's value, with the aim of minimizing $n_{10}(j)$. Assuming one bug in the system, O was proven to be optimal by Naish *et al.* [9]. Note that this optimality does not necessarily mean that it performs always better than other predictors [21].

As an attempt to relax the assumptions from O, Naish *et al.* also proposed the O^P fault predictor, which does not assign a negative score to every component j where $n_{01}(j) > 0$ holds [9]. In contrast, O^P scores components first based on their involvement in failing transactions and second on their involvement on passing transactions.

Ochiai, used in the context of fault localization by Abreu *et al.* [4], evaluates how similar a coverage matrix column $\mathcal{A}_j$ is from the error vector e [4]. It is a proxy for calculating the *cosine similarity* between two N-dimensional vectors.

The Tarantula predictor was proposed by Jones *et al.* [3] to assist fault localization using a visualization technique. The intuition behind this predictor is that components that are used often in failed executions, but seldom in passed executions, are more likely to be the root cause of observed failures.

C. Fix Cardinality

We now present a definition for the terms *fix*, *single-fault fix* and *multiple-fault fix* to be used throughout the paper.

Definition 4 (Fix)**.** The set of source code modifications that, when applied, eliminate a set of faults from the system.

Definition 5 (Single-Fault Fix)**.** A fix that eliminates one fault from the system.

Definition 6 (Multiple-Fault Fix)**.** A fix that eliminates more than one fault from the system.

To identify a fix as single-faulted, we check if all tests affected by the change — *i.e.*, that went from failing to passing — share at least one component modified by the fix. For that, we look at the minimal-cardinality *hitting-set* of tests affected by the change. If there are hitting sets of cardinality 1, it means that at least one of the components modified by the fix is active on every affected test. Note that even if there are tests that cover multiple components (*i.e.*, a system test) affected by change, we are assuming that there also is at least one test where the two unrelated components are not run together — thus breaking the hitting set, making it of cardinality > 1. Multiple-fault fixes are ones whose *hitting-sets* are of size greater than one.

D. Diagnosing Multiple Faults

While there are many flavors of fault predictors for spectrum-based fault localization — some of them described in Section II-B — they all end up assigning a one-dimensional score to each component j in the system. The fact is that these per-component analyses abstract away relevant information to properly score multiple-faulted subjects. This is also exacerbated in the O fault predictor, which is specifically designed to assume that there is only one fault affecting the system.

While there are more intricate approaches to fault localization that are able to produce accurate multiple-fault approaches, like model-based debugging [11] and spectrum-based reasoning [8], this comes at a cost — such techniques are computationally much more expensive and may require some partial modeling of the system under analysis.

DiGiuseppe *et al.* [22], while studying the influence of multiple faults on spectrum-based fault localization techniques, have actually found that at least some kinds of faults were localizable regardless of the presence of other faults. The authors state that, while the presence of more than one fault added noise to the ranking, such noise did not adversely affect the localizability of prominent types of faults.

To handle multiple faults, *debugging in parallel* was also proposed [12], [13]. This technique clusters test cases so that the resulting spectra each contain a different fault and therefore can be diagnosed using spectrum-based fault localization.

However, what remains to be seen is whether developers are actually faced with the task of having to diagnose and fix multiple bugs at once. Our intuition is that bugs are detected one-at-a-time when new fault-finding tests are added to the system. As such, we argue that the bug clustering from debugging in parallel approaches actually happens organically over the course of software development. Note that this does not mean that the system contains only one bug at a time, but rather that fixes are single-faulted: aside from the fault being fixed, all other faults in the system remain undetected.

III. Research Questions

In Section II we described several fault-localization predictors and stated that, as they yield a uni-dimensional ranking of fault likelihood, their accuracy might be impacted in multiple-fault scenarios. However, our hypothesis is that such multiple-faulted scenarios are not that frequent during development

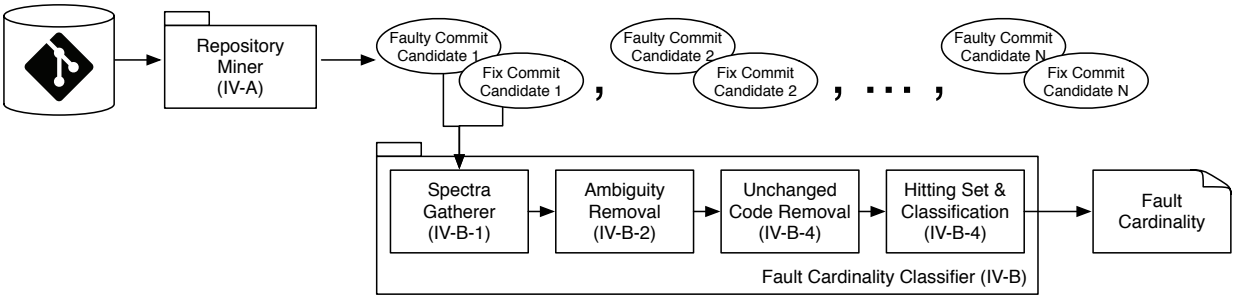


Fig. 2: Methodology for mining and classifying fixes.

because developers tend to fix faults soon after they are detected. This work aims to assess the prevalence of single-fault fixes in several real, open-source software projects.

Furthermore, for cases where multiple-faults are present in a system, we aim to quantify what is the decrease in diagnostic performance (if any) of using such fault-localization techniques to debug the system.

This work aims to address the following research questions:

RQ1: How prevalent are single-fault fixes in open-source projects?

RQ2: What is the effort to diagnose single-faults with *state-of-the-art* fault predictors?

RQ3: What is the impact on diagnostic performance when multiple-faults are considered?

RQ1 is concerned with the quantitative assessment of single-fault fixes and how their pervasiveness compares to that of multiple-fault fixes. In **RQ2**, we ask what is the diagnostic efficiency of current fault-localization approaches — most of which designed to pinpoint single-faults — when solely considering single-faulted scenarios. In **RQ3**, we shift our attention towards multiple-faulted scenarios, and ask what the diagnostic performance of fault-localization techniques is for these scenarios, with the aim of comparing against the single-fault baseline.

IV. Methodology for Fault Classification

This section details the methodology we followed for mining a project's repository, finding fixing commits, and labeling them as either single- or multiple-fault fixes. A diagram depicting the methodology is shown in Figure 2. Although we motivate our approach by mentioning spectrum-based fault localization, the methodology described in this section is completely separate from any diagnostic process.

A. Mining Fixing Commits

We employ a methodology for fault classification that involves access to the subject program's code repository, to enable the inspection of both the project's commit history and each *commit tree* (which represents the state of all checked-in files at a particular commit). We start by analyzing the latest commit in the default branch — which, in most workflows, is the `master` branch — and iteratively explore parent commits. This reverse-chronological exploration is able to handle most workflows enabled by advanced version control systems, such as branch merging, rebasing and commit *cherry-picking*.[2] However, note that the use of history-rewrite features like *commit squashing*[3] may influence the outcome of the fault cardinality classifier, as many commits are collapsed into one.

During our reverse-chronological analysis, we restore the working tree of the commit currently being explored and run the project's test-suite. If it is a passing suite, the commit is then considered as a fix candidate, and we advance to its parent commit, restoring its working tree. After that we run the fix candidate's suite — hence why our analysis is reverse chronological: so that the fixing commit's test-suite is run against an earlier commit. If the suite fails, we prompt the fault cardinality classifier, described in Section IV-B, to run. If, on the other hand, the suite passes, the commit's own test-suite is run to decide whether it should be the new fix candidate. This process repeats until all commits are explored.

B. Classifying Fault Cardinality of a Fixing Commit

We now describe the methodology for classifying the fault cardinality of any fixing commit discovered in Section IV-A. At this stage we execute test-suites at each commit under analysis and perform code coverage instrumentation. We have selected method-level granularity for the instrumentation thus methods are the units of our analysis. This way, fixes that only encompass one method are classified as single-faulted. Our classification methodology encompasses four steps:

1) Gathering Spectra: The first step in the fault classification process is to run the fixed version's test-suite against both fixed and faulty programs and gather their hit-spectra with methods as the component granularity. By faulty programs we mean programs compiled from source code in which the fixing set of commits was rolled back. When testing against the fixed version, we ensure that every test is passing. Since tests pass in the fixed version, we attribute any test failure observed when

[2] *Cherry-picking* refers to the act of applying the changes from a set of commits to the current branch.

[3] *Commit squashing* is the act of merging together a series of commits so they appear as one in the commit history.

testing the faulty program to the code changes between the two versions under test.

Figure 3 depicts an example hit-spectrum generated by the two test runs. Figure 3a shows the faulty version's spectrum and Figure 3b shows the fixed version's counterpart. Highlighted components denote elements from the Δ set — the set of components that were modified between the two versions under test.

$\mathcal{T}$	c_1	c_2	c_3	c_4	c_6	c_7	c_8	e
t_1	1	1	0	0	1	0	0	pass
t_2	0	1	1	0	1	1	0	fail
t_3	0	0	0	1	0	0	1	pass
t_4	1	0	1	0	0	1	0	fail
	Δ		Δ					

(a) Faulty version.

$\mathcal{T}$	c_1	c_2	c_3	c_4	c_6	c_7	c_8	e
t_1	1	1	1	0	1	0	0	pass
t_2	0	1	1	0	1	1	0	pass
t_3	1	0	0	1	0	0	1	pass
t_4	1	0	1	0	0	1	0	pass
	Δ		Δ					

(b) Fixed version.

Fig. 3: Spectra gathered when running test-suite from the fixed version. Δ denotes components changed by the fixing commit.

Note that, for the suite to run against the faulty version, the Δ set must not include any interface changes. If, on the other hand, there are compilation or runtime errors pertaining to an interface change, Δ is considered as containing changes to functionality and thus is discarded from analysis.

2) Ambiguity Removal: After gathering the faulty version of a spectrum, we perform an initial filtering step to remove ambiguous components, so that only one component from an ambiguity group is present. At the hit-spectrum level of abstraction, components can form an ambiguity group (also known as an equivalence class) if they always exhibit the same execution behavior, so it is not possible to distinguish between them. This inter-dependence means that these components will always need to be inspected together. Therefore, if a bug occurs in an ambiguity group, the group will be considered as faulty. An example of this filtering step is depicted in Figure 4. We consider that if any component in ambiguity group belongs to Δ, then the ambiguity group also belongs to Δ. This is so that the newly created ambiguity group component can be considered in the next steps of the analysis.

3) Unchanged Code Removal: The faulty version's spectrum shows test failures shows test failures not present in the fixed spectrum (*cf.* Figure 3). Recall that the test-suite is unchanged between the two versions, as described in our step 1, so we can attribute the cause of the erroneous behavior to a subset of components $\mathcal{C} \subseteq \Delta$ which is part of the code modified between the two versions under test.[4] All components not in Δ can therefore be safely exonerated from suspicion of

[4]This assumes test outcomes are deterministic.

$\mathcal{T}$	c_1	c_2	c_3	c_4	e
t_1	1	1	0	0	pass
t_2	0	1	1	0	fail
t_3	0	0	0	1	pass
t_4	1	0	1	0	fail
	Δ		Δ		

Fig. 4: Ambiguity group filtering step. Components from Figure 3a that exhibit the same behavior are grouped and collapsed into a single component.

containing the observed fault and are filtered out from the analysis, as shown in Figure 5.

$\mathcal{T}$	c_1	c_3	e
t_1	1	0	pass
t_2	0	1	fail
t_3	0	0	pass
t_4	1	1	fail

Fig. 5: Filtered from Figure 4 components not involved in Δ.

4) Hitting Set & Classification: The last filtering step to be performed is one that looks at failing tests from the faulty spectrum — namely t_2 and t_4 from our example — and keep them in the analysis. Passing tests — t_1 and t_3 — are discarded, as they do not reveal information about the faulty components.

The final, filtered spectrum, shown in Figure 6, is then submitted to minimal hitting set analysis [23], [24] so that we determine what (sets of) components are active on every failing row of the hit-spectrum. These sets are known as diagnostic candidates, since, when assumed faulty, can explain every observed error in the system. Each diagnosis candidate — a subset of all components — is valid if every failing test-case involves at least one component from the candidate. A candidate is *minimal* if removing any component from it makes it no longer a hitting set. We are only interested in minimal candidates, as they can subsume others of higher cardinality.

$\mathcal{T}$	c_1	c_3	e
t_2	0	1	fail
t_4	1	1	fail

Fig. 6: Spectrum depicting a single-fault after filtering passing tests.

If the minimal hitting set of the filtered spectrum yields solutions of cardinality 1, it means that there is at least one component that is involved every fault-revealing test. In Figure 6, component c_3 is active in every test, so we consider the fixing commit as being a *single-fault fix*.

Another example is depicted in Figure 7. In this filtered spectrum, the set of components $\{c_3, c_5\}$ has the minimal cardinality so that it explains every test failure. The fact that this spectrum's hitting set contains candidates of cardinality greater than 1 means that there is no one component that, when modified, causes all tests to pass. This is true because

no component is active on every failing test. Therefore, in this case, the fixing commit is labeled as a *multiple-fault fix*.

$\mathcal{T}$	c_1	c_3	c_5	e
t_2	0	1	0	fail
t_4	1	1	0	fail
t_5	0	0	1	fail

Fig. 7: Spectrum depicting a multiple-fault.

V. Empirical Study

To answer the research questions outlined in Section III, we conducted a large scale empirical study a large scale involving hundreds of open-source projects. The experiment entailed mining their code repositories and finding fault-fixing commits following the methodology described in Section IV. Afterward, each pinpointed fault was diagnosed using the fault-localization techniques described in Section II. This section describes our experimental setup and reports our findings.

A. Experimental Setup

The subjects of our study are open-source software projects originally gathered for a study on pull request distributed development on Github [25]. It encompasses over 6,000 publicly available code repositories for projects written in Java, Javascript, Python, Ruby and Scala. We have chosen this dataset due to its breadth of subjects and the fact that the vast majority of them contain test-cases — which are a requirement for spectrum-based analyses. The dataset was, however, filtered to fit the needs of our experiment. We have applied the following project filtering schemes:

1) We only consider the dataset's 1,288 Java projects. Projects written in other languages were discarded due to the fact that our tooling only handles Java source code (1,288 subjects out of 6,001).
2) Non-Apache Maven projects were discarded. Maven is a requisite for our analysis because we use one of its plugins to instrument code at runtime to obtain a test execution's program spectra. We also ensure that the `mvn compile` command terminates successfully and all dependencies can be resolved (701 subjects out of 1,288).
3) Projects should contain tests, otherwise fault-localization tools are unable to perform the analysis (279 subjects out of 701).

Out of all projects from the dataset, we end up considering 279 Java projects as our subjects.[5] On average, subject's test-suites were comprised of 596 tests.

The repository miner for classifying fixes as described in Section IV is available at https://github.com/aperez/single-fault-prevalence. The miner uses the Python library GitPython[6] to iterate throughout the repository's history. Gathering program spectra is done through the DDU Maven plugin,[7] a tool that calculates a diagnosability metric based on program spectra [26]. The tool shares the same internals for runtime instrumentation of Java programs as the GZoltar fault-localization tool. Minimal hitting set computation from the fault classifier is done using Abreu *et al.*'s Staccato algorithm [23].

Spectrum-based fault localization is performed at the method granularity. This is to match the ground truth generated by the repository (*i.e.*, which components are indeed faulty), so that diagnostic effort measurements can be computed.

B. Metrics Used

To assess diagnostic performance, we resort to the effort measurement (also known as wasted effort), commonly used in fault-localization research [27]. Since fault-localization techniques output a ranked list of components sorted by some fault predictor score, effort measures the average number of components to be inspected by following the ranked list until the real faulty component is reached:

$$\text{effort}(j) = \frac{|\{s_i | s_i > s_j\}| + \frac{|\{s_k | s_k = s_j\}|}{2}}{|\mathcal{C}|} \tag{2}$$

where s_c is the fault predictor score for a component c. The effort value is typically normalized by dividing the number of components in a system. An effort value of 0 indicates an ideal ranking where the faulty component is at the top and therefore no spurious code inspections will occur. Conversely, effort's lower bound value is 1. It states that the entire system will be inspected until the real fault is reached.

In the case of multiple-fault scenarios, the effort metric by itself is insufficient to judge diagnostic efficiency [27]. This is because more than one faulty component is scattered throughout the ranked list produced by fault-localization techniques. Given a set of k faulty components $\mathcal{F} = \{f_1, f_2, \cdots, f_k\}$, to better assess the diagnostic efficiency in these scenarios we provide three measurements:

1) First-fault effort, which is the effort required to reach the first faulty component:

$$\min \{\text{effort}(j) | j \in \mathcal{F}\} \tag{3}$$

2) Average-fault effort, an average of efforts to reach all faulty components:

$$\overline{\{\text{effort}(j) | j \in \mathcal{F}\}} \tag{4}$$

3) Worst-fault effort, the effort required to reach the last faulty component in the ranking:

$$\max \{\text{effort}(j) | j \in \mathcal{F}\} \tag{5}$$

We also compare the performance of each multiple-fault scenario to an artificially-crafted single-faulted equivalent. The artificial scenario is a proxy for a spectrum that only contains one component responsible for all erroneous behavior, aiding us to compare and contrast the outcome of fault-localization techniques between single- and multiple-fault versions of the

[5]The full list of experimental subjects is available at https://github.com/aperez/single-fault-prevalence.

[6]Available at https://pypi.python.org/pypi/GitPython.

[7]Available at https://github.com/aperez/ddu-maven-plugin.

same problem. For that, we merge all faulted components into one, as depicted in the example from Figures 8a and 8b. Figure 8a shows a program spectrum with its two faults highlighted — *i.e.*, components c_1 and c_3. Our merging strategy creates a new spectrum (Figure 8b) with all faulty components stripped and inserts a new component with the faulty components' coverage activity merged — this essentially amounts to performing a *bitwise or* among all faulty components' columns from the original spectrum. This way, we can judge what was the impact on diagnostic accuracy by measuring:

$$\Delta_{\text{effort}} = \text{effort} - \text{effort}_{\text{merged}} \tag{6}$$

Δ_{effort} values range from -1 to 1. A value of 1 means that the diagnostic efficiency is minimal in the multiple-fault scenario and maximal in its single-faulted equivalent. Conversely, Δ_{effort} = -1 states that efficiency is maximal for the original scenario and minimal for the single-faulted one. $\Delta_{\text{effort}} = 0$ means that both scenarios yield the same effort to diagnose.

$\mathcal{T}$	c_1	c_2	c_3	c_4	e
t_1	0	1	1	0	pass
t_2	1	0	0	0	fail
t_3	0	1	0	1	pass
t_4	0	0	1	0	fail

(a) Before faulty component merger.

$\mathcal{T}$	c_2	c_4	$c_{1,3}$	e
t_1	1	0	1	pass
t_2	0	0	1	fail
t_3	1	1	0	pass
t_4	0	0	1	fail

(b) After faulty component merger.

Fig. 8: Multiple-fault components merge strategy.

In the eventuality of a spectrum's minimal hitting set step described in Section IV-B4 producing more than one minimal-cardinality result, it means that the spectrum has more than one fault candidate — *i.e.*, there are multiple sets of components that can independently explain failing tests. Figure 9 provides an example scenario where the hitting set encompasses two fault candidates of cardinality 1 — c_1 and c_3. At the spectrum level of abstraction, one cannot distinguish the real fault set among the minimal-cardinality elements of the hitting set. To do so, one has to look at the source code from the fixing commit that provides the ground truth — and in most cases even then this is not enough, as one may need domain knowledge about the problem that the program is trying to solve to perform such code inspection. However, from a spectrum-based fault localization perspective, the fact is that any fault candidate could contain the fault. Since we are interested in the general case, we average the fault predictor values and effort scores for every scenario that has more than one minimal-cardinality set able to explain all observed errors. Note that this applies to both single-faulted and multiple-faulted scenarios.

$\mathcal{T}$	c_1	c_2	c_3	c_4	e
t_1	1	1	1	0	fail
t_2	1	0	1	0	fail
t_3	0	0	1	1	pass
t_4	0	1	0	1	pass

Fig. 9: Spectrum with multiple minimal-cardinality hitting sets.

C. Results

Out of the 279 subjects considered for evaluation, our classifier found fixing commits in 72 of them. What this figure tells us is not that 207 projects did not have any bug fixes, but rather that test-suites, when run against older versions of the code, do not produce any test-failure (although some still produce runtime errors, which are discarded as discussed in Section IV-B1). This means that no regressions are found throughout the projects' history. Also regarding these 207 projects, we can state that when developers find bugs in the code, they either (1) do not create a test-case exercising such a fault or (2) do not isolate their changes, adding code for new functionality to the same commit, along with code for new test-cases, causing it to no longer being labeled as a bug-fixing commit.

Overall, 12,417 commits were inspected, resulting in 1375 detected fixes (11% of inspected commits). Out of the detected fixes, 1135 of them (82.5%) were single-faulted, where one of the components modified by the fixing commit is sufficient to explain the faulty version's failing tests.

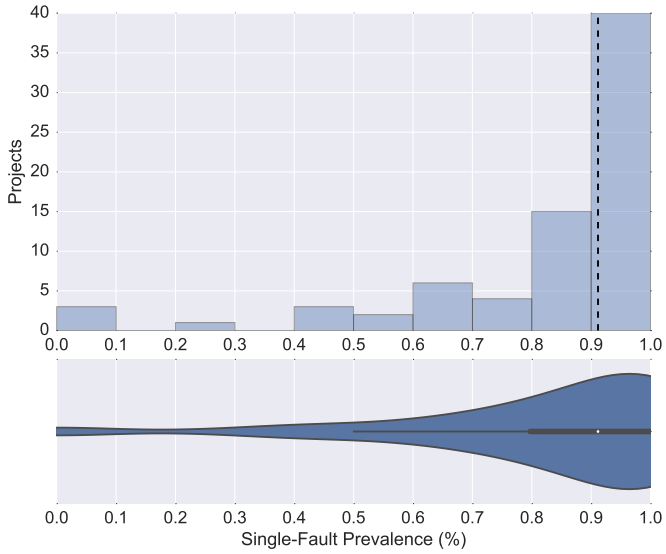
The histogram and violin plot from Figure 10a illustrate single-fault prevalence on different projects. For example, the prevalence of single faults ranges from 80 to 90% for 15 projects. The vertical dashed line in the plot indicates the median value of single-fault prevalence, 91.1%. Note that more than half of all projects considered show single-fault prevalence of 90% or more. Even considering the 25% percentile, the prevalence figure amounts to 79.6%. These prevalence results attest to the ubiquity of single-faults in open-sourced projects. Figure 10b shows fault cardinality for all detected fixes. It shows the quantity of fixes decaying exponentially with the fault cardinality.

Revisiting the first research question:

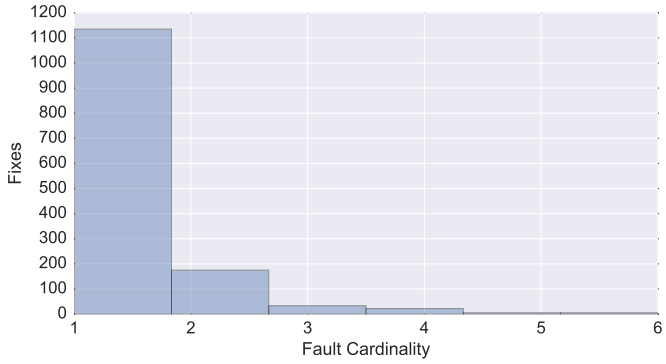
> **RQ1:** How prevalent are single-fault fixes in open-source projects?

Answer: We observed that 1135 fixes out of 1375 have eliminated a single fault from the system, yielding a single-fault prevalence of 82.5%.

We now shift our attention to the fixes classified as single-faulted to assess their diagnostic performance. For that, we show Figure 11, which is a cumulative plot of the effort required to detect the faults in our dataset when following the ranking generated by each fault predictor. The dashed vertical line represents an effort threshold of 0.05. We vi-



(a) Histogram and violin plot of single-fault prevalence. A violin plot is the combination of a box plot and a kernel density plot.



(b) Histogram of fault cardinality.

Fig. 10: Quantitative analysis of detected fixes.

sually show this threshold because one should not assume developers continue following the fault-localization ranking at this point, particularly when considering large codebases. This thresholding criterion is in agreement with the study by Parnin *et al.* [28], where developers were found to abandon the ranking if they inspected too many false positives; and the work of Nguyen *et al.* in program repair [14], that uses the predictor ranking as a set of clues to start the automated repair process, and places an explicit time-bounded threshold in the exploration of the ranking.

It is immediately apparent that the O fault predictor has the highest diagnostic efficiency, with over 90% of faults being at the top of their respective rankings (since their low effort value). O^P also fares comparably to O, diagnosing over 80% of faults with virtually zero wasted effort. In fact, at the exploration threshold, O and O^P manage to detect 95.2% and 92.5% respectively, attesting to their accuracy. Other fault predictors fare worse compared to both O and O^P. In fact, at the effort $= 0.05$ threshold highlighted in the vertical dashed line, other fault predictors' detection rate ranges from 57.2% to 79.5% of total faults.

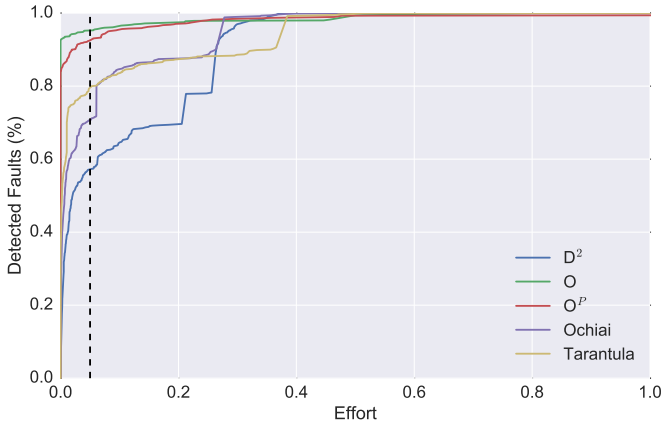


Fig. 11: Diagnostic effort throughout single-fault scenarios. Values at threshold – D^2: 57.2%, **O: 95.2%**, O^P: 92.5%, Ochiai: 70.6%, Tarantula: 79.5%.

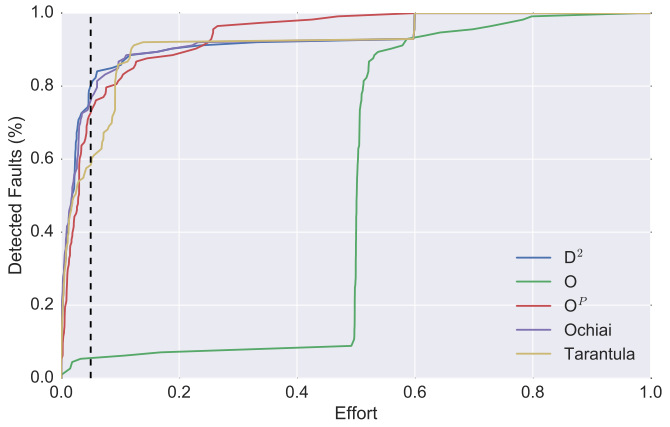
Revisiting the second research question:

> **RQ2:** What is the effort to diagnose single-faults with *state-of-the-art* fault predictors?

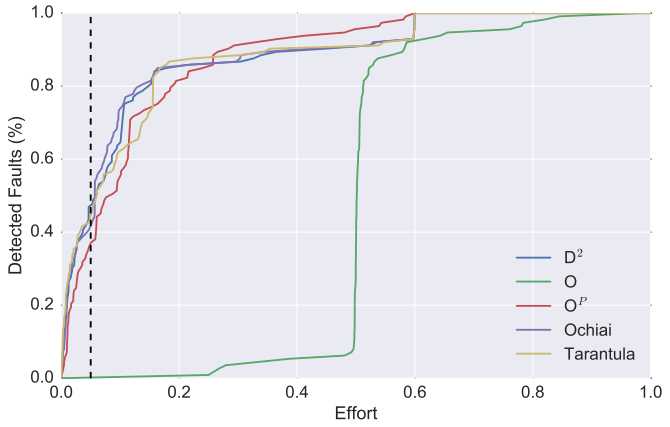
Answer: We show diagnostic for several fault predictors. One that fared consistently well was the O metric, with a fault detection rate amounting to over 90% while having virtually zero wasted effort.

Lastly, we look at multiple-fault scenarios. It is worth reminding the reader that a single effort measurement is insufficient to accurately portray diagnostic efficiency for multiple-faulted spectra. The effort metric measures the number of inspections required until the fault is reached. However, the erroneous behavior spans more than one component in these multiple-faulted scenarios. Hence, we introduce in Section V-A the notion of first-fault, average-fault and worst-fault efforts. Figures 12a to 12c plot such metrics for multiple-faulted scenarios.

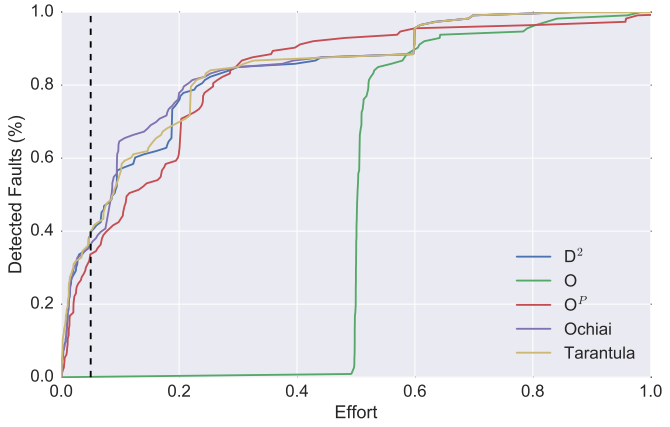
From the outset, we can notice that first-fault, average-fault and worst-fault efforts for the O fault predictor are very different from the remaining predictors. O detects a very low amount of components at low effort, an then there is a sudden jump in detection rate at effort's halfway point. In fact, this is to be expected, since according to O's definition, it attributes a score of -1 to every component in which $n_{01} > 0$. In other words, any component that is not active in every failing test, is given a negative score. However, in multiple-fault scenarios, there is rarely ever a faulty component active in all failing tests, meaning that according to O most components in the system will be scored with the same value, at which point locating the fault becomes essentially a random task. The other metrics produce fairly high fault-detection rates at low effort values to reach at least one of the faulty components (*i.e.*, considering the first-fault effort), with D^2, O^P and Ochiai having over 70%



(a) First-fault. Values at threshold – **D^2: 79.6%**, O: 3.5%, O^P: 72.6%, Ochiai: 75.2%, Tarantula: 58.4%.



(b) Average-fault. Values at threshold – **D^2: 46.9%**, O: 0%, O^P: 36.3%, Ochiai: 41.5%, Tarantula: 44.2%.



(c) Worst-fault. Values at threshold – **D^2: 39.8%**, O: 0%, O^P: 33.6%, Ochiai: 36.2%, Tarantula: 39.5%.

Fig. 12: Diagnostic effort throughout multiple-fault scenarios.

detection rate at our effort $= 0.05$ threshold. As to be expected, the detection rate decreases from the first-fault to the average-fault and from the average-fault to the worst-fault. Considering worst-fault effort (*i.e.*, the effort required to pinpoint all faults in the system), detection rates range from 33.6% using O^P to 39.8% using D^2.

TABLE II: Metric medians and statistical tests.

		Median Value	Median Effort	Median Δ_{Effort}	Wilcoxon Signed-rank
First Fault	**D^2**	1.00	0.02	0.00	$Z = 3059.0$ $p\text{-value} = 0.885$
	O	-1.00	0.50	0.50	$Z = 0.0$ $p\text{-value} = 4.07\times10^{-20}$
	O^P	3.01	0.03	0.02	$Z = 604.5$ $p\text{-value} = 2.85\times10^{-13}$
	Ochiai	0.56	0.02	-0.02	$Z = 1046.0$ $p\text{-value} = 6.06\times10^{-09}$
	Tarantula	0.99	0.02	-0.02	$Z = 2351.0$ $p\text{-value} = 3.64\times10^{-02}$
Average-Fault	**D^2**	0.96	0.05	0.04	$Z = 2183.0$ $p\text{-value} = 2.95\times10^{-03}$
	O	-1.00	0.50	0.50	$Z = 0.0$ $p\text{-value} = 2.78\times10^{-20}$
	O^P	2.50	0.07	0.07	$Z = 0.0$ $p\text{-value} = 2.78\times10^{-20}$
	Ochiai	0.42	0.05	0.00	$Z = 2901.0$ $p\text{-value} = 0.651$
	Tarantula	0.92	0.05	0.00	$Z = 2656.0$ $p\text{-value} = 0.105$
Worst-Fault	**D^2**	0.17	0.08	0.06	$Z = 2070.0$ $p\text{-value} = 9.79\times10^{-04}$
	O	-1.00	0.50	0.50	$Z = 0.0$ $p\text{-value} = 2.78\times10^{-20}$
	O^P	1.03	0.10	0.09	$Z = 0.0$ $p\text{-value} = 2.78\times10^{-20}$
	Ochiai	0.26	0.07	0.03	$Z = 1914.0$ $p\text{-value} = 1.05\times10^{-03}$
	Tarantula	0.85	0.07	0.03	$Z = 1950.0$ $p\text{-value} = 2.72\times10^{-04}$

Table II provides some additional information about the multiple-faulted scenarios. The table shows, for each fault predictor, its median value, median effort, and median Δ_{effort} — used for comparing against an equivalent single-faulted scenario, as described in Section V-A. A statistical test we performed but omitted from the table due to space constraints was the was the Shapiro-Wilk test for normality of effort data. The results tell us that the distributions are not normal, with confidence of 99%. That allowed us to perform a Wilcoxon signed-rank test to compare with the artificially-crafted equivalent single-faulted scenario, whose results are seen in the last column of Table II.

Given that the effort data is not normally distributed and that each observation is paired, we use the non-parametrical statistical hypothesis test Wilcoxon signed-rank. Our null-hypothesis is that the median difference between the two observations (*i.e.*, Δ_{effort}) is zero. We show the resulting Z statistic and p-value. With 99% confidence, we can refute the null-hypothesis in all scenarios but D^2 first-fault, Tarantula first-fault, Ochiai average-fault and Tarantula average-fault. In these cases, the effort values are comparable to their single-faulted counterparts. In cases where the null-hypothesis is refuted, only one yielded a negative Δ_{effort} — Ochiai first-fault — meaning that finding the first component out of the multiple components that comprise the fault was faster than finding the merged single-faulted component. All in all, we can say that, except for the O metric where Δ_{effort} has a big magnitude, the effort measurements are comparable

to their single-faulted counterparts when we consider the effort required to find one fault in the ranking. Diagnostic performance decreases when considering the effort required to find all faults in the system.

Revisiting the third research question:

> **RQ3:** What is the impact on diagnostic performance when multiple-faults are considered?

Answer: With the exception of the O fault predictor, which performs with random accuracy, the first-fault effort measurements of other fault predictors are comparable to the diagnostic effort for single-faulted equivalent scenarios. To diagnose all faults in a system, the fault predictors' accuracy decreases. Aside from O, the performance of other predictors when faced with multiple-fault scenarios is similar.

VI. Discussion

We discuss the practical implications of our findings, as well as outline the potential threats to their validity.

A. Practical Implications

Practical implications of this study are:

- We argue that our experimental results suggest a methodology to be followed when developers face failing test cases. As we have shown that there is a high likelihood that there is only one bug detected by failing tests, developers can try to find the fault by inspecting the ranking generated by the O^P fault predictor, since it produces near-optimal scores in the event of single-faults while still being usable in multiple-faulted scenarios (unlike O).
- Results suggest that closely monitoring the system as it develops (through, for instance, a continuous integration platform) and attempting to locate faults as soon as failures emerge will yield debugging tasks that require less wasted effort. This is because the likelihood of the fix being single-faulted is high when compared to only dealing with debugging tasks once there is a significant number of failing tests.
- Further research is needed to find whether there is a fault predictor that is closer to showing optimal accuracy when diagnosing multiple faults, exhibiting a Δ_{effort} that approaches zero for worst-fault scenarios.
- Effective automatic fault localization paves the way to other automatic techniques, such as automated program repair. Our experimental results yields insight into which technique will work best in practice. In particular, the prevalence of single-fault fixes suggest that the O^P fault predictor will yield near-optimal rankings as input to automatic repair techniques, while still providing some guidance in the event of multiple faults.

B. Threats to Validity

The main threat to validity of this study is related to external validity. When choosing the projects for our study, our aim was to opt for projects that resemble a general large-sized application being worked on by several people. To reduce selection bias and facilitate the comparison of our results, we decided to use the real-world scenarios described in the dataset gathered by Gousios *et al.* [25].

A potential threat to construct validity relates to our definition of what constitutes single-faulted and multiple-faulted fixes. Another threat to construct validity is our assumption that any interface change is result of a change in requirements and not the consequence of a fix. Lastly, we point out that history-rewrite features of modern version control systems can influence the outcome of the fault cardinality classifier. It can be the case that many single-faulted fixing commits collapsed into one large commit that is responsible for fixing multiple-faults by means of *commit squashing*.

The main threat to internal validity lies in the complexity of several of the tools used in our experiments, most notably our code instrumentation tool to retrieve spectra information.

VII. Conclusion

We study the prevalence of single-fault fixes in open-source Java projects, motivated by the fact that fault predictors (used by spectrum-based fault-localization approaches) can perform optimally in the event of a system being single-faulted. Our hypothesis is that while a software application can have many dormant bugs, these bugs are detected (and fixed) individually, thus constituting single-faulted events.

We describe an approach for mining software repositories in search for fixes — source code modifications that eliminate faults from the system — and for classifying said fixes according to the number of bugs they eliminate. The motivation behind creating such methodology is to study how debugging actually happens in practice and whether there is prevalence of single-fault fixes throughout the development of software.

We conducted an experiment with hundreds of open-source Java projects, mining their repositories and cataloguing the identified fixes. Overall, we have found 1375 fixes in over 70 projects. Out of all fixes, 82.5% were single-faulted (*i.e.*, only eliminate one bug from the system), indicating that single-faults are indeed prevalent among fixes. We have found that, for the detected single-faults, fault predictors O and O^P manage to achieve high diagnostic accuracy. For the remaining, multiple-faulted scenarios, average-fault and worst-fault diagnostic accuracy decayed slightly using most fault predictors. A glaring exception is with the O predictor that, because it assumes systems are single-faulted, yields an essentially random diagnostic performance.

Our experimental results suggest that using the O^P predictor is a sound methodology since it produces near-optimal results for diagnosing single-faults, while still achieving usable performance in multiple-faulted scenarios.

For future work, we aim to expand the scope of our analysis to cover other fault localization methodologies besides spectrum-based fault localization, such as mutant-based fault localization [29], [30], delta debugging [31] and model-based debugging [32], among others.

ACKNOWLEDGMENTS

This material is based upon work supported by Fundação para a Ciência e Tecnologia (FCT) under the scholarship No. SFRH/BD/95339/2013 and from the Brazilian research agencies CNPq (under grants 457756/2014-4 and 203981/2014-6) and FACEPE (under grant APQ-0402-1.03/13).

REFERENCES

[1] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey of software fault localization," *IEEE Transactions on Software Engineering*, 2016.

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

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

[4] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "An evaluation of similarity coefficients for software fault localization," in *12th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2006), 18-20 December, 2006, University of California, Riverside, USA*, 2006, pp. 39–46.

[5] Lucia, D. Lo, L. Jiang, F. Thung, and A. Budi, "Extended comprehensive study of association measures for fault localization," *Journal of Software: Evolution and Process*, vol. 26, no. 2, pp. 172–219, 2014.

[6] T.-D. B. Le, F. Thung, and D. Lo, "Theory and practice, do they match? a case with spectrum-based fault localization." in *ICSM*, 2013, pp. 380–383.

[7] T. Y. Chen, X. Xie, F. Kuo, and B. Xu, "A revisit of a theoretical analysis on spectrum-based fault localization," in *39th IEEE Annual Computer Software and Applications Conference, COMPSAC 2015, Taichung, Taiwan, July 1-5, 2015. Volume 1*, 2015, pp. 17–22.

[8] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "Spectrum-based multiple fault localization," in *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*, 2009, pp. 88–99.

[9] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, p. 11, 2011.

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

[11] W. Mayer and M. Stumptner, "Model-based debugging - state of the art and future challenges," *Electr. Notes Theor. Comput. Sci.*, vol. 174, no. 4, pp. 61–82, 2007.

[12] J. A. Jones, M. J. Harrold, and J. F. Bowring, "Debugging in parallel," in *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2007, London, UK, July 9-12, 2007*, 2007, pp. 16–26.

[13] W. Hogerle, F. Steimann, and M. Frenkel, "More debugging in parallel," in *25th IEEE International Symposium on Software Reliability Engineering, ISSRE 2014, Naples, Italy, November 3-6, 2014*, 2014, pp. 133–143.

[14] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: program repair via semantic analysis," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, 2013, pp. 772–781.

[15] M. Böhme and A. Roychoudhury, "Corebench: studying complexity of regression errors," in *International Symposium on Software Testing and Analysis, ISSTA '14, San Jose, CA, USA - July 21 - 26, 2014*, 2014, pp. 105–115.

[16] V. Dallmeier and T. Zimmermann, "Extraction of bug localization benchmarks from history," in *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, 2007, pp. 433–436.

[17] J. Sliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" in *Proceedings of the 2005 International Workshop on Mining Software Repositories, MSR 2005, Saint Louis, Missouri, USA, May 17, 2005*, 2005.

[18] A. Avižienis, J. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable Secure Computing*, vol. 1, no. 1, pp. 11–33, 2004.

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

[20] W. E. Wong, V. Debroy, Y. Li, and R. Gao, "Software fault localization using dstar (d*)," in *Sixth International Conference on Software Security and Reliability, SERE 2012, Gaithersburg, Maryland, USA, 20-22 June 2012*, 2012, pp. 21–30.

[21] S. Yoo, X. Xie, F.-C. Kuo, T. Y. Chen, and M. Harman, "No pot of gold at the end of program spectrum rainbow: Greatest risk evaluation formula does not exist," *RN/14/14*, 2014, University College London.

[22] N. DiGiuseppe and J. A. Jones, "On the influence of multiple faults on coverage-based fault localization," in *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, 2011, pp. 210–220.

[23] R. Abreu and A. J. C. van Gemund, "A low-cost approximate minimal hitting set algorithm and its application to model-based diagnosis," in *Eighth Symposium on Abstraction, Reformulation, and Approximation, SARA 2009, Lake Arrowhead, California, USA, 8-10 August 2009*, 2009.

[24] A. Feldman, G. M. Provan, and A. J. C. van Gemund, "Computing minimal diagnoses by greedy stochastic search," in *23rd AAAI Conference on Artificial Intelligence (AAAI)*, 2008, pp. 911–918.

[25] G. Gousios and A. Zaidman, "A dataset for pull-based development research," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014, 2014, pp. 368–371.

[26] A. Perez, R. Abreu, and A. van Deursen, "A test-suite diagnosability metric for spectrum-based fault localization approaches," in *39th International Conference on Software Engineering, ICSE*, 2017 (to appear).

[27] F. Steimann, M. Frenkel, and R. Abreu, "Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators," in *International Symposium on Software Testing and Analysis, ISSTA '13, Lugano, Switzerland, July 15-20, 2013*, 2013, pp. 314–324.

[28] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*, 2011, pp. 199–209.

[29] M. Papadakis and Y. L. Traon, "Using mutants to locate "unknown" faults," in *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST*, 2012, pp. 691–700.

[30] L. Zhang, L. Zhang, and S. Khurshid, "Injecting mechanical faults to localize developer faults for evolving software," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA*, 2013, pp. 765–784.

[31] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 183–200, 2002.

[32] W. Mayer and M. Stumptner, "Evaluating models for model-based debugging," in *23rd IEEE/ACM International Conference on Automated Software Engineering, ASE*, 2008, pp. 128–137.