Theory and Practice, Do They Match?
A Case With Spectrum-Based Fault Localization

Tien-Duy B. Le, Ferdian Thung, and David Lo
School of Information Systems
Singapore Management University, Singapore
{btdle.2012,ferdianthung,davidlo}@smu.edu.sg

Abstract—Spectrum-based fault localization refers to the process of identifying program units that are buggy from two sets of execution traces: normal traces and faulty traces. These approaches use statistical formulas to measure the suspiciousness of program units based on the execution traces. There have been many spectrum-based fault localization approaches proposing various formulas in the literature. Two of the best performing and well-known ones are Tarantula and Ochiai. Recently, Xie et al. [18] find that theoretically, under certain assumptions, two families of spectrum-based fault localization formulas outperform all other formulas including those of Tarantula and Ochiai. In this work, we empirically validate Xie et al.’s findings by comparing the performance of the theoretically best formulas against popular approaches on a dataset containing 199 buggy versions of 10 programs. Our empirical study finds that Ochiai and Tarantula statistically significantly outperform 3 out of 5 theoretically best fault localization techniques. For the remaining two, Ochiai also outperforms them, albeit not statistically significantly. This happens because an assumption in Xie et al.’s work is not satisfied in many fault localization settings.

I. INTRODUCTION

In software systems, bugs are unavoidable. Many bugs are regularly found and reported to the developers. The amount of bugs to be fixed is often much larger compared to the size of the development team [3]. To tackle this problem, researchers have developed automated approaches to help developers in fixing bugs. These automated approaches include many fault localization techniques proposed in the literature [15], [9], [1], [19], [13], [14]. The goal of fault localization is to localize a bug to local regions of the source code. Thus, rather than the whole program, developers only need to investigate a much smaller part of the program. This would significantly reduce the amount of time needed to find the buggy program elements and fix the bug.

One large family of fault localization techniques is Spectrum-Based Fault Localization (SBFL) techniques [15], [9], [1], [19], [13]. SBFL techniques analyze program spectra, which are program traces collected during the execution of a program, to correlate failures (i.e., faulty execution traces) with program elements (e.g., lines, basic blocks) that are responsible for them. Various SBFL techniques use various formulas to assign suspiciousness scores to program elements. Program elements are then ranked based on their suspiciousness scores. The resulting ranked list is then given to developers to help them find the root cause of failures. Two well-known SBFL techniques are Tarantula [9] and Ochiai [1].

Recently, Xie et al. [18] have theoretically investigated many SBFL formulas. Their study has shown that SBFL formulas can be grouped into families (or equivalence classes). Within each family, the formulas have the same effectiveness to localize bugs under certain assumptions. Also, they have created a partial order which shows which families of SBFL formulas are better than others. At the top of the partial order are 2 families of SBFL formulas named ER1 and ER5 which contain in total 5 SBFL formulas. Xie et al. have theoretically proven that the 5 SBFL formulas can outperform Tarantula’s and Ochiai’s SBFL formulas. However, these SBFL formulas have not been empirically compared with one another on actual failures and programs.

In this study, we want to inspect the applicability of the theoretically best SBFL formulas to localize faults in standard SBFL benchmark dataset. Xie et al. theoretical analysis assumes that the test coverage level is 100%. This assumption is likely not to hold for many fault localization settings. Thus, there is a need for an empirical study to demonstrate whether these theoretically best formulas could outperform popular formulas in many fault localization settings.

In this empirical study, we use 199 buggy versions of 10 programs: NanoXML, XML-Security, Space, and the 7 programs from the Siemens test suite [8]. We want to answer the following research questions:

RQ 1 How effective are the popular and theoretically best SBFL formulas?
RQ 2 Could the theoretically best SBFL formulas outperform the popular formulas?
RQ 3 Is the assumption considered by Xie et al. [18] satisfied in many fault localization settings?

Our empirical study demonstrates that among the 7 SBFL formulas (5 theoretically best and 2 popular formulas), Ochiai’s SBFL formula performs the best. Using it, on average, developers only need to inspect 21.02% of the source code. Among the theoretically best formulas, the best percentage is 21.09%. According to the Wilcoxon signed rank test [16], Ochiai’s SBFL formula statistically significantly outperforms 3 out of the 5 theoretically best SBFL formulas.

The following are our contributions:

1) We empirically evaluate the effectiveness of the theoretically best SBFL formulas against popular ones (i.e.,...
TABLE I

<table>
<thead>
<tr>
<th></th>
<th>$e$ Executed</th>
<th>$e$ Not Executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Passed</td>
<td>$n_s(e)$</td>
<td>$n_s(\bar{e})$</td>
</tr>
<tr>
<td>Test Failed</td>
<td>$n_f(e)$</td>
<td>$n_f(\bar{e})$</td>
</tr>
</tbody>
</table>

The notation $\bar{e}$ means $e$ is not executed. $n_s(e)$ denotes the number of successful test cases that execute $e$, $n_f(e)$ denotes the number of failing test cases that execute $e$, $n_s(\bar{e})$ denotes the number of successful test cases that do not execute $e$, and $n_f(\bar{e})$ denotes the number of failing test cases that do not execute $e$.

B. Popular Approaches: Tarantula and Ochiai

Many approaches have been proposed to compute the suspiciousness scores of program elements [9], [1], [13], [18]. Tarantula [9] and Ochiai [1] are among the most popular approaches. Using the notations in Table I, Tarantula’s SBFL formula, which assigns a suspiciousness score to a program element $e$, is defined as follows:

$$\text{Tarantula}(e) = \frac{n_f(e)}{n_s(e) + n_f(e)}$$

where $n_f = n_f(e) + n_f(\bar{e})$ and $n_s = n_s(e) + n_s(\bar{e})$.

Ochiai’s SBFL formula is defined as follows:

$$\text{Ochiai}(e) = \frac{n_f(e)}{\sqrt{n_f(n_f(e) + n_s(e))}}$$

C. Theoretically Best SBFL Formulas

Xie et al. [18] have compared 30 SBFL formulas and theoretically prove that two families of theoretically best SBFL formulas outperform others, including those of popular approaches like Tarantula and Ochiai. They refer to these two families as $ER_1$ and $ER_5$. $ER_1$ has two members: $ER_1^a$ and $ER_1^b$. $ER_5$ has three members: $ER_5^a$, $ER_5^b$, and $ER_5^c$. Using the notations in Table I, the following are the definitions of those formulas which assign a suspiciousness score to a program element $e$:

$$ER_1^a(e) = \begin{cases} 
-1, & \text{if } n_f(e) < n_f \\
 n_s - n_s(e), & \text{if } n_f(e) = n_f 
\end{cases}$$

$$ER_1^b(e) = n_f(e) - \frac{n_s(e)}{n_s(e) + n_s(\bar{e}) + 1}$$

$$ER_5^a(e) = n_f(e)$$

$$ER_5^b(e) = \frac{n_f(e)}{n_f(e) + n_f(\bar{e}) + n_s(e) + n_s(\bar{e})}$$

$$ER_5^c(e) = \begin{cases} 
0, & \text{if } n_f(e) < n_f \\
 1, & \text{if } n_f(e) = n_f 
\end{cases}$$

III. METHODOLOGY

In this section, we first describe the dataset that we use to investigate the effectiveness of SBFL approaches. Next, we describe how we collect traces from this dataset. We then describe how we measure effectiveness.

A. Dataset

Our dataset consists of buggy versions of 10 programs: NanoXML, XML-Security, Space, and the 7 programs from the Siemens test suite [8]. NanoXML is a Java library for XML parsing. XML-Security is a Java library for encryption and digital signature. Space is an Array Definition Language (ADL) interpreter written in C. Siemens test suite is a suite created by Siemens for research in test coverage adequacy. NanoXML, XML-Security, and Space are downloaded from the Software Infrastructure Repository (SIR) [5]. For NanoXML and XML-Security, we exclude faulty versions that do not have failing
In order to evaluate the effectiveness of a SBFL formula, we count the percentage of executable code that needs to be inspected to find the first faulty program element. We have also investigated the reason why the theoretically best SBFL formulas cannot outperform popular techniques despite the theoretical analysis given in [18]. We investigate the code coverage of the buggy versions of the 10 programs. We find that out of the 199 buggy versions, for 135 of them, the code coverage is not 100%. The average code coverage for the 199 buggy versions is 84.97%. This highlights the reason why the theoretical findings in [18] does not hold for many fault localization settings.

D. Threats to Validity

Threats to internal validity refers to errors or experimental bias. We have double checked our code and implementation of the formulas. Still there could be errors that we do not notice.

Threats to external validity refers to the generalizability of our findings. We have analyzed 199 buggy versions from 10 programs. These programs and buggy versions have been used to evaluate many past SBFL studies [1], [9], [15], [12], [10].

### IV. Experiments & Analysis

In this section, based on the methodology described in Section III, we describe the answers to the 3 research questions that we listed in Section I.

#### A. RQ1: Effectiveness of SBFL Formulas

The effectiveness of the various SBFL formulas are shown in Table III. The average percentage of program elements to be inspected to find the first faulty program element are 23.37%, 21.02%, 33.34%, 21.09%, 43.04%, 43.04%, and 54.95% for Tarantula, Ochiai, ER1a, ER1b, ER5a, ER5b, and ER5c respectively. Assuming that e2 is the faulty program element, in the worst case, developers need to inspect 3 program elements to reach the faulty program element. Thus, the EXAM score for this example is \( \frac{3}{4} = 75\% \).

#### B. RQ2: Popular vs. Best Approaches

From Table III, we notice that Ochiai has the lowest EXAM score. The EXAM score of Tarantula is also lower than 4 out of the 5 theoretically best SBFL formulas. We have also performed non-parametric statistical tests, i.e., Wilcoxon signed rank tests [16], with a significance level of 0.05. We find that the EXAM scores of Ochiai are statistically significantly better than those of ER5a, ER5b, ER5c.

#### C. RQ3: Validity of Assumptions

We have also investigated the reason why the theoretically best SBFL formulas cannot outperform popular techniques despite the theoretical analysis given in [18]. We investigate the code coverage of the buggy versions of the 10 programs. We find that out of the 199 buggy versions, for 135 of them, the code coverage is not 100%. The average code coverage for the 199 buggy versions is 84.97%. This highlights the reason why the theoretical findings in [18] does not hold for many fault localization settings.

### TABLE II

<table>
<thead>
<tr>
<th>Dataset</th>
<th>LOC</th>
<th>Language</th>
<th># Faulty</th>
<th># Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_token</td>
<td>478</td>
<td>C</td>
<td>5</td>
<td>4130</td>
</tr>
<tr>
<td>print_token 2</td>
<td>399</td>
<td>C</td>
<td>10</td>
<td>4115</td>
</tr>
<tr>
<td>replace</td>
<td>312</td>
<td>C</td>
<td>51</td>
<td>5542</td>
</tr>
<tr>
<td>schedule</td>
<td>292</td>
<td>C</td>
<td>9</td>
<td>2050</td>
</tr>
<tr>
<td>schedule 2</td>
<td>301</td>
<td>C</td>
<td>9</td>
<td>2710</td>
</tr>
<tr>
<td>jcas</td>
<td>141</td>
<td>C</td>
<td>56</td>
<td>1608</td>
</tr>
<tr>
<td>joc_info</td>
<td>340</td>
<td>C</td>
<td>19</td>
<td>1031</td>
</tr>
<tr>
<td>space</td>
<td>6218</td>
<td>C</td>
<td>55</td>
<td>17355</td>
</tr>
<tr>
<td>NanoXML v1</td>
<td>3,997</td>
<td>Java</td>
<td>6</td>
<td>214</td>
</tr>
<tr>
<td>NanoXML v2</td>
<td>4,007</td>
<td>Java</td>
<td>7</td>
<td>214</td>
</tr>
<tr>
<td>NanoXML v3</td>
<td>4,008</td>
<td>Java</td>
<td>8</td>
<td>215</td>
</tr>
<tr>
<td>NanoXML v4</td>
<td>4,782</td>
<td>Java</td>
<td>8</td>
<td>216</td>
</tr>
<tr>
<td>XML security v1</td>
<td>21,613</td>
<td>Java</td>
<td>6</td>
<td>92</td>
</tr>
<tr>
<td>XML security v2</td>
<td>22,118</td>
<td>Java</td>
<td>6</td>
<td>94</td>
</tr>
<tr>
<td>XML security v3</td>
<td>19,893</td>
<td>Java</td>
<td>4</td>
<td>84</td>
</tr>
</tbody>
</table>

### TABLE III

<table>
<thead>
<tr>
<th>Technique</th>
<th>Average % Inspected</th>
<th>Standard Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tarantula</td>
<td>23.37%</td>
<td>25.44%</td>
</tr>
<tr>
<td>Ochiai</td>
<td>21.02%</td>
<td>21.96%</td>
</tr>
<tr>
<td>ER1a</td>
<td>33.34%</td>
<td>35.22%</td>
</tr>
<tr>
<td>ER1b</td>
<td>21.09%</td>
<td>19.48%</td>
</tr>
<tr>
<td>ER5a</td>
<td>43.04%</td>
<td>19.63%</td>
</tr>
<tr>
<td>ER5b</td>
<td>43.04%</td>
<td>19.63%</td>
</tr>
<tr>
<td>ER5c</td>
<td>54.95%</td>
<td>26.83%</td>
</tr>
</tbody>
</table>
In the future, we plan to reduce this threat to validity further by investigating more bugs from more software systems.

Threats to construct validity refers to the suitability of our evaluation measure. We have used the EXAM score which is used to evaluate many past SBFL studies [17], [1]. The study by Xie et al. [18] also theoretically compares the performance of many SBFL formulas using the EXAM score.

V. RELATED WORK

In the following paragraphs, we first highlight some SBFL studies. Next, we also briefly discuss other fault localization approaches that do not rely on program spectrum. Due to the space constraint, the survey here is by no means complete.

SBFL. Many SBFL approaches have been proposed in the literature [15], [9], [1], [19], [13]. All these techniques analyze program spectra which are logs of execution traces generated when a target program is run. Zeller proposes a technique named Delta Debugging which finds the minimum state difference that causes a failure to be generated [2]. Renieris and Reiss compare a faulty execution with the nearest correct execution to find suspicious program elements [15]. Jones and Harrold propose an SBFL technique named Tarantula which uses a formula to compute suspiciousness of program elements based on the assumptions that program elements executed more by faulty executions rather than by correct executions are more likely to be faulty [9]. Abreu et al. propose another SBFL technique named Ochiai that uses another formula to compute suspiciousness of program elements [1]. Lucia et al. investigate the effectiveness of many association measures for fault localization [13]. Gong et al. propose an interactive SBFL approach that takes incremental user input into consideration [7]. Gong et al. also propose another SBFL approach that reduces the number of test cases with oracles [6]. Cheng et al. mine graph-based signatures that highlight suspicious program elements by analyzing program spectra [4]. Duy and Lo propose a classification-based approach that predicts whether an SBFL technique would be effective for a particular fault localization task [11]. Xie et al. theoretically analyze many SBFL formulas including Tarantula and Ochiai and show that two families of SBFL formulas (ER1 and ER5) could outperform the others if a number of assumptions hold [18]. In this work, we compare the effectiveness of the theoretically best formulas presented in Xie et al.’s work with Tarantula and Ochiai using a standard SBFL benchmark dataset.

Other Fault Localization Approaches. Aside from SBFL, a number of past papers have also proposed model-based fault localization techniques which often use formal models and employ expensive logic reasoning, e.g., [14]. This limits the applicability of this family of fault localization approaches especially on large complicated programs. In this work, we only consider SBFL approaches.

VI. CONCLUSION AND FUTURE WORK

We have conducted an empirical evaluation of various SBFL techniques on 199 buggy versions of NanoXML, XML-Security, Space, and the 7 programs from the Siemens test suite. We compare the performance of 5 theoretically best SBFL formulas presented by Xie et al. [18] with popular SBFL formulas (Tarantula and Ochiai). We find that Ochiai’s SBFL formula outperforms all, while Tarantula’s SBFL formula outperforms four theoretically best SBFL formulas. For three out of the five theoretically best formulas, Ochiai and Tarantula SBFL formulas statistically significantly outperform them. We highlight that the assumption made by Xie et al. is not valid for many settings. For many programs, even though with a large number of test cases, the code coverage is not 100%. A relatively small reduction in test coverage can significantly affect the performance of the theoretically best SBFL formulas.

As a future work, we plan to perform a more in-depth study on how coverage levels and other factors affect the effectiveness of various SBFL formulas. We are also interested in theoretically analyzing the performance of SBFL formulas under a more relaxed assumption (i.e., less than 100% coverage). Furthermore, we want to reduce the threat to external validity by investigating more programs and buggy versions.

REFERENCES