2013 IEEE International Conference on Software Maintenance

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* outperforms 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 the 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.,

1063-6773/13 $26.00 © 2013 IEEE
DOI 10.1109/ICSM.2013.52

TABLE I
RAW STATISTICS FOR SBFL

	e Executed	e Not Executed
Test Passed	$n_s(e)$	$n_s(\bar{e})$
Test Failed	$n_f(e)$	$n_f(\bar{e})$

Tarantula's and Ochiai's SBFL formulas). We find that Ochiai's SBFL formula statistically significantly outperforms 3 out of the 5 theoretically best SBFL formulas. For the remaining two, Ochiai's SBFL formula performs better, although the differences are not statistically significant.

2) We highlight that the assumption made by Xie et al. in their theoretical analysis, that the code coverage level is 100%, is not satisfied in many fault localization settings which affects the performance of the *theoretically best* SBFL formulas.

The following is the structure of the paper. In Section II, we introduce SBFL and highlight Ochiai, Tarantula, and the two families of theoretically best SBFL formulas [18]. In Section III, we describe our empirical study methodology. In Section IV, we present the answers to the research questions. We discuss related work in Section V. We conclude and mention future work in Section VI.

II. BACKGROUND

In this section, we first succinctly introduce SBFL. Next, we describe the formulas used in two popular approaches namely Tarantula [9] and Ochiai [1]. We then highlight the formulas demonstrated by Xie et al. [18] to be theoretically the best.

A. SBFL in a Nutshell

SBFL is a technique to localize a bug to certain parts of the program by utilizing program spectra collected from software testing and the result of the tests (*pass* or *fail*). Program spectra, which is a record of which program elements are executed for each test case, can be collected at different levels of granularity (e.g., lines, basic blocks, methods, components, etc.). In this paper, we consider the basic block level granularity (i.e., each basic block is a program element). SBFL requires a set of test cases where at least one of the test cases results in a faulty execution (i.e., the test case fails). For each program element e, SBFL computes the raw statistics shown in Table I.

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.

Based on the statistics, a suspiciousness score for each program element e is computed. The higher the score is, the more suspicious the program element is. Thus, a ranked list of program elements sorted by their suspiciousness scores is returned. The list can then be investigated by developers, starting from the most suspicious program element.

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:

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

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:

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

C. Theoretically Best SBFL Formulas

Xie et al. [18] have compared 30 SBFL formulas and theoretically prove that two families of SBFL formulas outperform others, including those of popular approaches like Tarantula and Ochiai. They refer to these two families as $ER1$ and $ER5$. $ER1$ has two members: $ER1^a$ and $ER1^b$. $ER5$ has three members: $ER5^a$, $ER5^b$, and $ER5^c$. Using the notations in Table I, the following are the definitions of those formulas which assign a suspiciousness score to a program element e:

$$ER1^a(e) = \begin{cases} -1, & \text{if } n_f(e) < n_f \\ n_s - n_s(e), & \text{if } n_f(e) = n_f \end{cases}$$

$$ER1^b(e) = n_f(e) - \frac{n_s(e)}{n_s(e) + n_s(\bar{e}) + 1}$$

$$ER5^a(e) = n_f(e)$$

$$ER5^b(e) = \frac{n_f(e)}{n_f(e) + n_f(\bar{e}) + n_s(e) + n_s(\bar{e})}$$

$$ER5^c(e) = \begin{cases} 0, & \text{if } n_f(e) < n_f \\ 1, & \text{if } n_f(e) = n_f \end{cases}$$

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

TABLE II
DATASET DESCRIPTIONS: NAME, LINES OF CODE, PROG. LANGUAGE, NUMBER OF FAULTY VERSIONS, AND NUMBER OF TEST CASES.

Dataset	LOC	Language	# Faulty	# Tests
print_token	478	C	5	4130
print_token2	399	C	10	4115
replace	512	C	31	5542
schedule	292	C	9	2650
schedule2	301	C	9	2710
tcas	141	C	36	1608
tot_info	440	C	19	1051
space	6,218	C	35	13,585
NanoXML v1	3,497	Java	6	214
NanoXML v2	4,007	Java	7	214
NanoXML v3	4,608	Java	8	216
NanoXML v5	4,782	Java	8	216
XML security v1	21,613	Java	6	92
XML security v2	22,318	Java	6	94
XML security v3	19,895	Java	4	84

test cases or the faulty program elements are not executed by any test case. For Siemens test suite, we exclude versions where the fault is located in the variable declaration. This is done since our instrumentation cannot reach it. Each faulty version contains one bug that may span across multiple program elements (i.e. basic blocks). Table II shows some statistics of our dataset. These programs and buggy versions have been used as a benchmark dataset used to evaluate many past SBFL studies [1], [9], [15], [12], [13], [10].

B. Collecting Execution Traces

Execution traces are collected at the basic block level. Each basic block in a program is instrumented by a "print" statement such that we can detect whether a particular basic block is executed when a test is run. For each basic block, a matrix shown in Table I is maintained. Based on the status of a test (i.e., pass or fail) and whether a basic block is executed when the test is run, the corresponding element in the matrix is updated. For example, if the test fails and basic block A is executed, then $n_f(A)$ is increased by 1. The matrices for the other basic blocks in the program are also updated accordingly. This process is repeated for each test. In the end, each basic block will have a matrix reflecting its execution profile.

C. Measuring Effectiveness of SBFL Approaches

In order to evaluate the effectiveness of a SBFL formula, we count the percentage of executable code that needs to be inspected to reach the first faulty program element. In the case that many program elements share the same suspiciousness score with the faulty program element, we assign the worst rank to the faulty program element (i.e. the faulty program element has the largest rank among all program elements with the same score). This measure is referred to as the *EXAM* score [17]. The following is the formula for calculating the *EXAM* score:

$$\textit{EXAM score} = \frac{\text{Rank of the first faulty program element}}{\text{Total number of executable program elements}}$$

The lower the *EXAM* score, the better is the performance of a SBFL formula. To illustrate *EXAM* score computation, consider four program elements e_1, e_2, e_3, and e_4 in a program with suspiciousness scores of 1.0, 0.75, 0.75, and 0.5 respectively. Assuming that e_2 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 $\frac{3}{4} = 75\%$.

TABLE III
EFFECTIVENESS OF THE SBFL FORMULAS

Technique	Average % Inspected	Standard Dev.
Tarantula	23.37%	23.44%
Ochiai	21.02%	21.96%
$ER1^a$	33.34%	35.22%
$ER1^b$	21.09%	19.48%
$ER5^a$	43.04%	19.63%
$ER5^b$	43.04%	19.63%
$ER5^c$	54.95%	26.83%

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, $ER1^a$, $ER1^b$, $ER5^a$, $ER5^b$, and $ER5^c$ respectively.

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 $ER5^a$, $ER5^b$, $ER5^c$.

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.

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],

[13], [10]. 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

[1] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the Accuracy of Spectrum-based Fault Localization," in *TAICPART-MUTATION*, 2007.

[2] Andreas Zeller, "Yesterday, my program worked. today, it does not. why?" in *ESEC / SIGSOFT FSE*, 1999, pp. 253–267.

[3] J. Anvik, L. Hiew, and G. C. Murphy, "Coping with an open bug repository," in *ETX*, 2005, pp. 35–39.

[4] H. Cheng, D. Lo, Y. Zhou, X. Wang, and X. Yan, "Identifying bug signatures using discriminative graph mining," in *ISSTA*, 2009.

[5] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact." *Empirical Software Engineering: An International Journal*, vol. 10, no. 4, pp. 405–435, 2005.

[6] L. Gong, D. Lo, L. Jiang, and H. Zhang, "Diversity maximization speedup for fault localization," in *ASE*, 2012.

[7] ——, "Interactive fault localization leveraging simple user feedback," in *ICSM*, 2012.

[8] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand, "Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria," in *Proc. of ICSE*, 1994, pp. 191–200.

[9] J. Jones and M. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *ASE*, 2005.

[10] M. Jose and R. Majumdar, "Cause clue clauses: error localization using maximum satisfiability," in *PLDI*, 2011, pp. 437–446.

[11] T.-D. B. Le and D. Lo, "Will fault localization work for these failures? an automated approach to predict effectiveness of fault localization tools," in *ICSM*, 2013.

[12] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "SOBER: Statistical model-based bug localization," in *ESEC/FSE*, 2005.

[13] Lucia, D. Lo, L. Jiang, and A. Budi, "Comprehensive evaluation of association measures for fault localization," in *ICSM*, 2010.

[14] W. Mayer and M. Stumptner, "Model-Based Debugging – State of the Art And Future Challenges," *ENTCS*, vol. 174, no. 4, 2007.

[15] M. Renieris and S. Reiss, "Fault localization with nearest neighbor queries," in *ASE*, 2003, pp. 141–154.

[16] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[17] W. E. Wong, V. Debroy, and D. Xu, "Towards better fault localization: A crosstab-based statistical approach," *IEEE Transactions on Systems, Man, and Cybernetics, Part C*, vol. 42, no. 3, pp. 378–396, 2012.

[18] X. Xie, T. Chen, F.-C. Kuo, and B. Xu, "A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization," *TOSEM (to appear)*, 2013.

[19] A. Zeller, "Isolating cause-effect chains from computer programs," in *FSE*, 2002, pp. 1–10.