Are Faults Localizable?

Lucia, Ferdian Thung, David Lo, and Lingxiao Jiang
School of Information Systems, Singapore Management University
{*lucia.2009,ferdianthung,davidlo,lxjiang*}*@smu.edu.sg*

***Abstract*—Many fault localization techniques have been proposed to facilitate debugging activities. Most of them attempt to *pinpoint* the location of faults (i.e., localize faults) based on a set of failing and correct executions and expect debuggers to investigate a certain number of located program elements to find faults. These techniques thus assume that faults are localizable, i.e., only one or a few lines of code that are close to one another are responsible for each fault. However, in reality, are faults localizable? In this work, we investigate hundreds of real faults in several software systems, and find that many faults may not be localizable to a few lines of code and these include faults with high severity level.**

***Keywords*-Fault Localization; Fault Locality; Bug Severity**

I. Introduction

Bugs are one major contributor to high software cost, and many automated debugging techniques have been proposed to reduce the cost of debugging. One kind of such techniques is fault localization that aims to pinpoint program elements responsible for a bug (i.e., the root causes, or a fault, of a bug). Many of these techniques analyze program spectra (i.e., a set of profiles of both correct and failed executions), with the goal of locating likely faulty program elements [8], [12], [18], [19].

Fault localization often assumes that faults are localizable, i.e., a fault is confined to one or a few lines of code that are close to each other in a software system. Most fault localization techniques would rank program elements in terms of their suspiciousness (i.e., the likelihood of a program element to be faulty), and expect developers to traverse this list of program elements in order and be able to decide whether an element is a fault by just inspecting that element. Past studies on fault localization often use faults that are injected to only one or a few locations, making the evaluation of the techniques biased.

The question is whether it is indeed the case that faults are confined to a few lines of code in real systems. *Are faults localizable?* This research question has important implications if faults turn out to be non-localizable. We may then need to re-consider the applicability of fault localization and design new approaches to aid developers in debugging non-localizable faults.

In this paper, we perform an empirical study by analyzing software from a public bug repository—iBugs [5] and the JIRA repository of Lucene. We consider hundreds of real bugs in three real systems: AspectJ, Rhino, and Lucene, and investigate how localized or spread-out the locations of faulty program elements are.

The contributions of this work are as follows:

- We highlight an important research question on whether faults are localizable in real software.
- We present an empirical study on three Java programs and note that many faults are not localized.
- We analyze whether severe faults are localizable.

The paper is organized as follows. Section II presents our dataset. Section III presents the setup of our study along with important manual cleanup of the dataset, the various locality definitions, and the research questions. Section IV presents our findings. We present related work in Section V and conclude with future work in Section VI.

II. Dataset

We analyze the locality of faults in two Java programs (Rhino and AspectJ) from iBugs repository [5] and a third Java program Lucene collected from JIRA [1] by another research team at UC Davis. Rhino is a Javascript interpreter written in Java with code size of about 49kLOC. There are 32 buggy versions of Rhino in iBugs. AspectJ is a compiler for aspect-oriented programming in Java with code size of about 75kLOC, and iBugs contains 350 of its buggy versions. Lucene is a text search engine library with code size of about 88kLOC (version 2.9).

The iBugs repository stores both pre-fix versions that contain bugs (buggy versions) and the corresponding post-fix versions where the bugs are fixed. Each of the buggy versions is assumed to contain one bug that may span across multiple lines in multiple files. Information about each fix is also provided based on the *diff* between the pre-fix and post-fix versions, e.g. the numbers of changed lines, changed methods, and changed files, and the severity level of the bug. Similar information based on the *diff* between bugs and corresponding fixes is also available for Lucene.

III. Empirical Study Setup

A. Extracting Faults from Changes

In this paper, we are concerned with program elements that are *responsible*, or are the root causes of a bug. The information from our datasets is not directly usable because the *diff* between a pre-fix and post-fix version may not correspond to root causes due to various reasons. First, the changed lines in the post-fix versions are often the *treatment*

of a fault and may not be the root causes themselves. For example, a piece of code may be moved around to make it easier to implement the actual fix. Second, not all changed lines are meant for fixing the bug. For example, previous studies (e.g., [10]) show that some changes are non-essential, such as changing indentation, adding comments, and code refactoring that do not change the behavior of a program.

To recover actual root causes from all changes so as to make our study more accurate, we perform a manual investigation on all bugs in our dataset. For each bug, we look at all of the changes between its pre-fix and post-fix versions and the surrounding code of the changes to decide which are the actual faulty lines in the pre-fix version, excluding changes related formatting, variable renaming, method renaming, refactoring, optimization, etc. We also exclude bugs that are explicitly marked as enhancement in the datasets, that do not contain any severity information, and that only have faults in test code or comments. When we deem there is ambiguity about where the actual faulty lines are (e.g., adding an if statement that may return a method), we conservatively treat such faults localizable by marking only one or few lines as faulty. We end up with 374 bugs with various numbers of manually labeled faulty lines: 32 for Rhino, 290 for AspectJ, and 52 for Lucene.

B. Definitions of Locality

In this paper, we define the *locality* of a fault based on involved faulty program elements. We consider program elements at three levels of granularity, including lines, methods, and files, and also consider the spatial distances among the program elements. The following is a list of locality definitions that we use in our empirical evaluation.

[D1] Considering a line of code as a program element, we define the locality L of a buggy version v as follows: $L_{D1}(v) =$ the number of faulty lines.

[D2] Considering a method as a program element, we define the locality L of a buggy version v as follows: $L_{D2}(v) =$ the number of faulty methods.

[D3] Considering a file as a program element, we define the locality L of a buggy version v as follows: $L_{D3}(v) =$ the number of faulty files.

[D4] Considering the spatial distances among the faulty lines, we define the locality L of a buggy version v based on the number of faulty files n_f and the gaps among the faulty lines in every file $G_{line}(file)$: $L_{D4}(v) = (\Sigma_{file} G_{line}(file) + n_f - 1) \times n_f$, where $G_{line}(file)$ is the distance between the first and last faulty lines in the file.

C. Research Questions

Based on the locality definitions, we consider the following research questions:

RQ1 Are faults localizable? How many faults could be localized to a few program elements?

RQ2 Are the most severe faults localizable?

IV. Results and Discussion

A. RQ1: Are Faults Localizable?

In this paper, we evaluate how localized are the faults in term of the number of faulty lines, methods, and files. Figures 1, 2, and 3 show the proportion of faults that are localizable up to a certain number of faulty lines, methods, and files respectively. Each figure shows the results for Rhino, AspectJ, Lucene, and the overall dataset. The detailed results are given in Tables I, II, and III.

Considering the number of faulty lines, faults in Rhino, AspectJ, and Lucene could span up to 957, 103, and 594 lines respectively. However, not many faults involve more than 50 lines—only 22%, 3%, and 8% for Rhino, AspectJ, and Lucene respectively. Figure 1 shows the proportion of faults that span across 1 to 50 lines of code. 22% of Rhino's faults, 37% of AspectJ's faults, and 19% of Lucene's faults involve one line. Overall, 33% of all faults involve one line. Considering 10 lines of code, 59% of Rhino's faults, 86% of AspectJ's faults, 83% of Lucene's faults, and 83% of all faults involve at most 10 lines. Therefore, most faults are localized within 10 lines of code.

Considering the number of faulty methods, faults in Rhino, AspectJ, and Lucene could span up to 296, 78, and 217 methods respectively. However, not many faults involve more than 20 methods—only 25%, 2%, 13% for Rhino, AspectJ, and Lucene respectively. Figure 2 shows the proportion of faults that span across 1 to 20 methods. 28% of Rhino's faults, 49% of AspectJ's faults, and 29% of Lucene's faults involve one method. Overall 44% of the faults involve one method. Also, 83% of the faults involve at most six methods.

Considering the number of faulty files, faults in Rhino, AspectJ, and Lucene could span up to 14, 56, and 33 files respectively. However, not many faults involve more than 10 files—only 3%, 1%, 6% for Rhino, AspectJ, and Lucene respectively. Figure 3 shows the proportion of faults that span across 1 to 10 files. 76% of Rhino's faults, 77% of AspectJ's faults, and 65% of Lucene's faults involve one file. Overall, 73% of the faults involve one file. Also, 88% of the faults involve at most two files.

Figure 4 shows the proportion of faults that have 1 to 1,000 gap locality scores. We notice that most faults have rather big gap locality across the three programs.

B. RQ2: Are the Most Severe Faults Localizable?

We evaluate the relationship between localizable faults and their severity levels. There are six severity levels in the AspectJ bugs that we analyze: blocker, critical, major, minor, normal, and trivial (from high to low). As for Rhino, there are only two levels: major and normal. For Lucene, there are four levels: blocker, major, minor, trivial. Table IV, V, and VI show the severity levels when faults reside in one line, less than ten lines, one method, and one file for

Table I
NUMBER & PERCENTAGE OF FAULTS (IN PARENTHESES) COVERING DIFFERENT NUMBER OF FAULTY LINES, METHODS, AND FILES FOR RHINO

Locality	Lines	Methods	Files
1	7 (22%)	9 (28%)	25 (76%)
2	3 (9%)	5 (16%)	3 (10%)
3	2 (6%)	4 (13%)	0 (0%)
4	0 (0%)	1 (3%)	2 (7%)
5	3 (9%)	1 (3%)	0 (0%)
6	0 (0%)	0 (0%)	0 (0%)
7	2 (6%)	0 (0%)	1 (3%)
8	1 (3%)	0 (0%)	0 (0%)
9	1 (3%)	1 (3%)	0 (0%)
10	0 (0%)	1 (3%)	0 (0%)

Table II
NUMBER AND PERCENTAGE OF FAULTS (IN PARENTHESES) COVERING DIFFERENT NUMBER OF LINES, METHODS, AND JAVA FILES FOR ASPECTJ

Locality	Lines	Methods	Files
1	106 (37%)	139 (48%)	213 (77%)
2	49 (17%)	49 (17%)	45 (16%)
3	32 (11%)	26 (9%)	15 (5%)
4	13 (4%)	15 (5%)	8 (3%)
5	16 (6%)	8 (3%)	3 (1%)
6	8 (3%)	10 (3%)	4 (1%)
7	9 (3%)	11 (4%)	0 (0%)
8	4 (1%)	8 (2%)	0 (0%)
9	5 (2%)	3 (1%)	0 (0%)
10	6 (2%)	4 (1%)	0 (0%)

Table III
NUMBER AND PERCENTAGE OF FAULTS (IN PARENTHESES) COVERING DIFFERENT NUMBER OF LINES, METHODS, AND JAVA FILES FOR LUCENE

Locality	Lines	Methods	Files
1	10 (19%)	15(29%)	34(65%)
2	8 (15%)	6(12%)	10(19%)
3	6 (12%)	5 (10%)	1(2%)
4	4 (8%)	7(13%)	1(2%)
5	2 (4%)	4(8%)	1 (2%)
6	3 (6%)	4(8%)	0(0%)
7	6 (12%)	2(4%)	1(2%)
8	1 (2%)	1(2%)	1 (2%)
9	1 (2%)	0(0%)	0 (0%)
10	2 (4%)	0(0%)	0 (0%)

Table IV
NUMBERS AND PERCENTAGES OF FAULTS FOR DIFFERENT SEVERITY LEVELS IN RHINO WHEN FAULTS ARE WITHIN 1 LINE, 10 LINES, 1 METHOD, OR 1 FILE

Bug severity	1 Line	10 Lines	1 Method	1 File
Major	0 (0%)	1 (50%)	1 (50%)	2 (100%)
Normal	7 (23%)	18 (60%)	8 (30%)	23 (85%)

Table V
NUMBERS & PERCENTAGES OF FAULTS FOR DIFFERENT SEVERITY LEVELS IN ASPECTJ WHEN FAULTS ARE WITHIN 1 LINE, 10 LINES, 1 METHOD, OR 1 FILE

Bug severity	1 Line	10 Lines	1 Method	1 File
Blocker	3 (43%)	7 (100%)	4 (57%)	6 (86%)
Critical	8 (33%)	19 (79%)	11 (46%)	16 (67%)
Major	12 (40%)	26 (87%)	15 (50%)	22 (73%)
Minor	6 (43%)	11 (79%)	5 (36%)	11 (79%)
Normal	75 (35%)	183 (86%)	102 (48%)	156 (73%)
Trivial	2 (100%)	2 (100%)	2 (100%)	2 (100%)

Table VI
NUMBERS AND PERCENTAGES OF FAULTS FOR DIFFERENT SEVERITY LEVELS IN LUCENE WHEN FAULTS ARE WITHIN 1 LINE, 10 LINES, 1 METHOD, OR 1 FILE

Bug severity	1 Line	10 Lines	1 Method	1 File
Blocker	1(33%)	2 (67%)	2(67%)	2(67%)
Major	5(20%)	18(72%)	7 (28%)	18(72%)
Minor	3(14%)	17(81%)	5(24%)	13(62%)
Trivial	1(50%)	2 (100%)	1(50%)	1(50%)

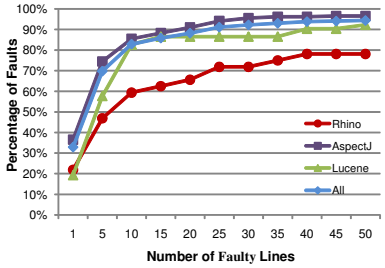

Figure 1. Proportion of faults versus line locality.

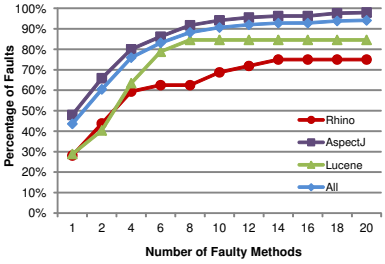

Figure 2. Proportion of faults versus method locality.

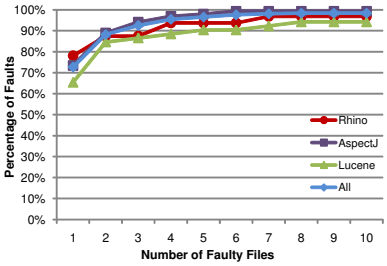

Figure 3. Proportion of faults versus file locality.

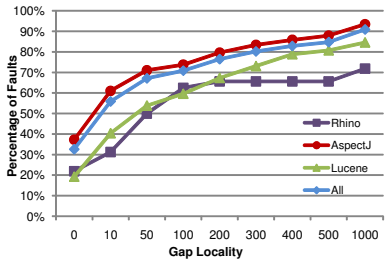

Figure 4. Proportion of faults versus gap locality.

Rhino, AspectJ, and Lucene respectively. For Rhino, 0%, 50%, 50%, and 100% of the major faults are localizable to one line of code, ten lines of code, one method, and one file, respectively. For AspectJ, 43%, 100%, 57%, and 86%, of the blocker faults are localizable to one line of code, ten lines of code, one method, and one file, respectively. For Lucene, 33%, 67%, 67%, and 67% of the blocker faults are localizable to one line of code, ten lines of code, one method, and one file, respectively. Overall, considering the most severe bugs in the category (i.e., blocker for Lucene and AspectJ, and major for Rhino), 33%, 83%, 58%, and 83% of them are localizable to one line of code, ten lines of code, one method, and one file, respectively, which means many of the most severe faults are not localizable.

C. Threats to Validity

Threat to external validity refers to the generalizability of our findings. We only analyze three real Java programs and may not be generalizable to other programs written in different languages. Due to lack of data, we do not carry out statistical significance test on the correlation between bug severity and fault locality. We leave this as our future work. Threat to internal validity is related to the manual investigation on what the actual faulty lines are, and possible implementation errors in the scripts to investigate the bugs.

V. Related Work

1) Fault Localization. There are many studies on fault localization [8], [12], [18], [19]. Renieris and Reiss contrast a failed execution to its nearest correct execution to find most suspicious program elements [18]. Zeller and Hildebrandt propose Delta Debugging that automatically isolates failure-inducing inputs [19]. Jones et al. propose Tarantula that ranks program elements based on the proportion of failed and correct executions that go through the elements [8]. Various other suspiciousness measures have also been proposed [2], [12], [13]. Artzi et al. propose Apollo to locate faults in web applications [3]. Recent techniques also extract bug signatures, but they often take exponential time and the signatures only cover a few lines of code [4], [7].

Most of these studies assume faults to be localizable (i.e., they could be pinpointed to one or a few lines of code). In this study, we empirically evaluate if this assumption holds.

There are also studies that investigate cases where multiple faults exist at the same time (e.g., [9]). Herzig and Zeller "untangle" changes in a commit that are unrelated with each other [6]. In this study, we assume changes in one commit deal with one bug only, and empirically analyze how widespread or localized the faults are.

2) Empirical Studies on Bugs. Pan et al. analyze patterns of bug fixes and classify bug fixes to different categories [15]. Related studies on fault characterization have also been performed by Ostrand and Weyuker [14], Perry and Stieg [17], and Leszak et al. [11]. In this study, we perform an orthogonal study investigating fault locality. Parnin and Orso perform user studies and found that the current fault localization techniques may not always be sufficient [16]. Our work analyzes the bugs themselves without user evaluation, but the results also bear a similar implication.

VI. Conclusion & Future Work

In this work, we perform a preliminary study for the question if bugs are localizable. We analyze hundreds of bugs and their fixes from three software systems, AspectJ, Rhino, and Lucene, and manually extract faults from changes. We find that (1) 67% and 56% faults are *not* localizable within one line and one method respectively, and that (2) 67% and 42% of the most severe faults (i.e., "blocker" in AspectJ and Lucene, and "major" in Rhino) are *not* localizable within one line and one method respectively. Thus, fault localization techniques may need be improved to report root causes that involve multiple lines or methods.

In the future, we plan to analyze more software systems and plan to build a system that could automatically recover faults from their treatments recorded in source control repositories and bug tracking systems. We also plan to investigate the faults that are not localizable and design new approaches that can help debuggers to fix those faults.

Acknowledgement

We thank Foyzur Rahman and Prem Devanbu for providing us a database containing the Lucene bugs with the associated fixes. We also thank the anonymous MSR reviewers for their valuable comments.

References

[1] "https://issues.apache.org/jira/browse/LUCENE."

[2] R. Abreu, "Spectrum-based fault localization in embedded software." Ph.D. dissertation, Delft University of Technology, 2009.

[3] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Practical fault localization for dynamic web applications," in *ICSE*, 2010.

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

[5] V. Dallmeier and T. Zimmermann, "Extraction of bug localization benchmarks from history," in *ASE*, 2007, pp. 433–436.

[6] K. Herzig and A. Zeller, "Untangling changes," September 2011.

[7] H.-Y. Hsu, J. A. Jones, and A. Orso, "Rapid: Identifying bug signatures to support debugging activities," in *ASE*, 2008, pp. 439–442.

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

[9] J. A. Jones, M. J. Harrold, and J. F. Bowring, "Debugging in parallel," in *ISSTA*, 2007.

[10] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories," in *ICSE*, 2011, pp. 351–360.

[11] M. Leszak, D. E. Perry, and D. Stoll, "A case study in root cause defect analysis," in *ICSE*, 2000, pp. 428–437.

[12] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug isolation via remote program sampling," in *PLDI*, 2003, pp. 141–154.

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

[14] T. J. Ostrand and E. J. Weyuker, "Collecting and categorizing software error data in an industrial environment," *Journal of Systems and Software*, vol. 4, no. 4, 1984.

[15] K. Pan, S. Kim, and E. J. W. Jr., "Toward an understanding of bug fix patterns," *Empirical Software Engineering*, vol. 14, no. 3, pp. 286–315, 2009.

[16] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *ISSTA*, 2011, pp. 199–209.

[17] D. E. Perry and C. S. Stieg, "Software faults in evolving a large, real-time system: a case study," in *ESEC*, 1993, pp. 48–67.

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

[19] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE TSE*, vol. 28, pp. 183–200, 2002.