Evaluating the Usefulness of IR-Based Fault Localization Techniques

Qianqian Wang
Georgia Institute of Technology, USA
q.wang@gatech.edu

Chris Parnin
North Carolina State University, USA
cjparnin@ncsu.edu

Alessandro Orso
Georgia Institute of Technology, USA
orso@cc.gatech.edu

ABSTRACT

Software debugging is tedious and time consuming. To reduce the manual effort needed for debugging, researchers have proposed a considerable number of techniques to automate the process of fault localization; in particular, techniques based on information retrieval (IR) have drawn increased attention in recent years. Although reportedly effective, these techniques have some potential limitations that may affect their performance. First, their effectiveness is likely to depend heavily on the quality of the bug reports; unfortunately, high-quality bug reports that contain rich information are not always available. Second, these techniques have not been evaluated through studies that involve actual developers, which is less than ideal, as purely analytical evaluations can hardly show the actual usefulness of debugging techniques. The goal of this work is to evaluate the usefulness of IR-based techniques in real-world scenarios. Our investigation shows that bug reports do not always contain rich information, and that low-quality bug reports can considerably affect the effectiveness of these techniques. Our research also shows, through a user study, that high-quality bug reports benefit developers just as much as they benefit IR-based techniques. In fact, the information provided by IR-based techniques when operating on high-quality reports is only helpful to developers in a limited number of cases. And even in these cases, such information only helps developers get to the faulty file quickly, but does not help them in their most time consuming task: understanding and fixing the bug within that file.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: [Testing and Debugging]

General Terms

Experimentation

Keywords

Fault localization, information retrieval, user studies

1. INTRODUCTION

Debugging is an important and extremely expensive activity that encompasses several tasks: fault localization, fault understanding, and fault removal. To reduce the manual effort involved in debugging, researchers have proposed a number of techniques. In particular, there is a rich body of work in the area of fault localization, and a great deal of research has been performed on techniques that aim to help developers locate bugs automatically. Most of these fault localization techniques are *spectra-based*—they analyze the dynamic information collected from program executions to locate faults (*e.g.,* [7, 11, 12]). In this context, a significant amount of effort has been devoted to finding better formulas that can result in a more precise localization. Although these improvements were reportedly effective, a recent study [24] showed that there is no best formula, and pursuing a perfect solution in therefore pointless. In addition, spectra-based techniques require a large number of both passing and failing executions, which are rarely available in practice.

To address these issues, and further improve automated debugging, researchers have proposed the use of information retrieval (IR) techniques to identify files that are likely to contain the fault(s) responsible for a failure (*e.g.,* [13,20,25]). The basic idea behind these techniques is to identify such files based on their lexical similarity with the content of the bug reports. Specifically, these approaches treat a bug report as a query and retrieve the source files that are most related to that query using a model extracted from the complete set of source files. The intuition behind these approaches is that, if a source file contains many words that are similar or identical to words in the bug report, it is very likely that such file is closely related to the corresponding bug. Although these techniques have been shown to be at least as effective as spectra-based fault localization techniques [18], there are several issues to be addressed before they can be widely used in real-world scenarios.

One important issue with IR-based approaches is the assumption that the bug reports provided by the users can work well as queries. Whether this assumption is satisfied depends on the type of software considered, the characteristics of the bug, and the background of the users reporting the bug. Another problem with these approaches is the coarse-grained nature of the results they produce. Most IR-based approaches provide results at the file level, which can still leave developers with a large amount of code to examine. In addition, and as shown in a previous study from two of the authors, providing results as a ranked list can hinder the usefulness of the approach [17]. In general, these techniques have never been evaluated through studies that involve actual developers, so there is limited evidence of their actual usefulness.

To address these issues, we performed two studies: one analytical and one using human subjects. In the analytical study, we identified several open source projects frequently used in previous work and performed a comprehensive evaluation of IR-based fault localization techniques on these projects. The results of this study confirm that IR-based techniques are more likely to produce good results when the provided bug reports contain rich information, and in particular, program entity names involved in the failure. An examination of 10,000 bug reports from Bugzilla shows, however, that only a small number of bug reports contain such information. In our user study, conversely, we investigated the usefulness of the fault localization results produced by an IR-based technique, by comparing the performance of a set of developers when performing a set of debugging tasks with and without the help of an IR-based technique. The results of this second study show that high-quality reports—those for which IR-based techniques work well—are in most cases good enough to guide the developers to the problematic files. Even in the few cases in which the IR-based technique made a difference, it still did not help the developers find the bug in the problematic files, which is the task that consumed most of the developers' time.

This paper makes the following contributions:

- A thorough investigation of real-world software systems and bug reports for those systems that reveals what information a bug report should contain for an IR-based technique to produce effective fault localization results.
- A user study involving developers with professional experience that investigates how IR-based fault localization techniques can affect the performance of developers in their debugging tasks.
- An in-depth analysis of the study results and a discussion of how these results may help further research in IR-based fault localization techniques and debugging in general.

2. BACKGROUND

In this section, we introduce information retrieval and explain how it works for fault localization.

2.1 Information Retrieval

Information Retrieval [14] is the activity of identifying relevant information in a collection of resources based on a query expressed as a set of keywords. For an IR system, the inputs are usually the query of interest and a corpus, consisting of a large collection of documents, whereas the output is a subset of documents related to the query, ranked by relevance (computed based on textual similarity).

A frequently used indexing statistic is *tf-idf* (term frequency-inverse document frequency) [14], which indicates the importance of a word for a document. The more frequently a word occurs in one document (higher *tf*), and the less frequently it occurs in other documents (higher *idf*), the more important it is for that document.

Queries and documents are represented in a suitable model once they are indexed. The basic representation models include, among others, the Vector Space Model (VSM) [21], Unigram Models (UM) [15], Latent Semantic Analysis (LSA) [4], and Latent Dirichlet Allocation (LDA) [1].

Following is a brief description of the models:

- **UM:** A probability model in which the probability of each word occurring in documents is treated independently. To avoid instances where the occurrence of a word in previous documents is 0, UM is often smoothed by assigning a non-zero probability, creating a Smoothed Unigram Model (SUM).
- **VSM:** An algebraic model that represents queries and documents as a vector whose elements are term-specific weights, such as tf-idf.
- **LSA:** A topic-based model that uses singular value decomposition (SVD) to represent the relationship between documents and their containing terms with a set of topics related to the documents and terms.
- **LDA:** Another topic-based model in which each document belongs to several topics with a certain probability distribution calculated with a Dirichlet prior.

2.2 IR-Based Fault Localization

When software users encounter a bug, they usually report it with a description of what happened or other related information, to help developers investigate the reason for and find the location of the bug in the source code. Since the goal of fault localization is to retrieve source code related to a bug, it can be modeled as an information retrieval problem. The input of the model is a bug report (the query) and the source files (the corpus), and the expected output is a source file (or a method) related to the bug described in the report, which is equivalent to the relevant documents retrieved in IR.

Because of the existence of programming language keywords (*e.g., double, if, else*) and identifiers consisting of concatenation of multiple words (*e.g., filterFile, canBuild, isLeap*), an extra preprocessing step is always performed when bug reports and source files are indexed. Together with normal IR preprocessing, this step includes text normalization (*e.g.,* removing punctuation, converting case), stop-word (including keyword) removal, and word splitting and stemming (converting words to their root forms). After preprocessing, a set of tokens are generated to be used as terms to index bug reports and documents. The similarity is then computed using the models mentioned in 2.1 to determine the relevance of each source file to the bug report. The most relevant source files are presented to the developers because, intuitively, the more relevant a source file is to a bug report, the more likely it is to contain the bug.

The effectiveness of IR-based fault localization is often evaluated using the following metrics [20]:

- Rank of Retrieved Files, which indicates how many files need to be examined to find the bug.
- Mean Reciprocal Rank (MRR), which is a statistic for evaluating a process producing a list of responses to a query.
- Mean Average Precision (MAP), which evaluates information retrieval models when a query may have multiple relevant documents.

Although some IR-based techniques have been reported to be effective based on these metrics [18, 20, 25], we believe that these earlier results are potentially problematic for several reasons. First, as previously reported [9], studying only reports for (already) localized bugs can substantially bias fault localization results, as it may consider only high-quality bug reports. The second issue is the assumption that all the files changed as part of a bug fix are responsible for a reported bug, and the faulty file is located as long as one of the changed files is located. Finally, IR-based techniques are never evaluated with real developers. In this paper, we try to answer the question of what information in particular affects the fault localization results and to find out the severity of the problems by an analytical study and a user study that explores how IR-based fault localization helps developers in debugging real-world bugs.

3. HYPOTHESES AND RESEARCH QUESTIONS

3.1 Hypotheses

The following hypotheses guided our study design:

Hypothesis 1: Not all bug reports can be used to locate bugs using information retrieval models. Only those with specific information can give acceptable fault localization results.

The quality of a bug report determines the quality of the result of IR-based fault localization techniques.

Hypothesis 2: Intuitively, if a bug report contains rich information, it will be easier for the developer to understand the failure and subsequently find the bug.

Developers who debug based on bug reports that contain richer information will be more effective at locating bugs than developers who debug with less information.

Hypothesis 3: Given a perfect ranked list of files, with the faulty file first, developers are likely to find the bug faster and more accurately than if no list were given.

Developers who are provided with a ranked list of potentially suspicious files will locate bugs more effectively than developers without such a list.

3.2 Research Questions

To test the above hypotheses, we formulated several research questions regarding the relationship between bug report quality and fault localization results, as well as how these results affect developers' debugging behavior.

Research Question 1: *What information in a bug report tends to produce good fault localization results?*

Our first research question asks what makes a good bug report in terms of producing good fault localization results. Previous research [9] has found that over 50% of the studied bug reports explicitly contained file locations, and these bug reports had significantly better fault localization performance. However, such research did not examine other items, such as method names or the context of the match. For example, if a file location is provided in a stack trace, then it might be one of many possible locations to inspect, and it might be significantly harder for a developer to find. To answer this question, we investigated previously studied benchmark programs to categorize information contained in bug reports and try to determine whether the information affects the IR-based fault localization techniques.

Research Question 2: *How do developers take advantage of existing information in the bug reports?*

After identifying the factors that affect fault localization results, our second research question tries to answer whether the same factors affect developers' debugging performance as well.

Table 1: Benchmark programs considered.

Project	Description	#Bugs	#Source Files
Aspectj	An aspect oriented programming extension	286	6k
SWT	An open source widget toolkit for Java	98	0.5k
ZXing	A barcode image processing library for Android	20	0.4k
Jodatime	A replacement for Java Date and Time classes	9	0.2k

Research Question 3: *Will developers behave and perform differently when they use an IR-based debugging tool than when they do not?*

Our third research question investigates comprehensively how the IR-based debugging tool affects developers' debugging behavior.

4. ANALYTICAL STUDY

To answer the first research question, we performed a comprehensive analytical study on the benchmarks used by previous researchers to evaluate the effectiveness of IR-based fault localization techniques.

4.1 Dataset

In our study, we selected four open source programs and obtained the associated bug reports. The projects are listed in Table 1. To obtain ranking information, we used the results generated by BugLocator [25]. We chose BugLocator because it was built based on VSM, the foundation of many state-of-the-art IR-based fault localization techniques. BugLocator revised VSM by taking into account the influence of file length and integrating the similarity between a new bug and previously fixed bugs. It has been shown that BugLocator can outperform several other IR-based techniques. Most importantly, the authors of BugLocator made both their tool and dataset available and many researchers have adopted the same dataset for evaluating their own techniques.

Before addressing RQ1 and assessing what information generates good fault localization results, it is important to understand what good fault localization results are. Previous evaluations of IR-based fault localization techniques used an optimistic approach, in which the evaluation metric for a result consisted of the *best* rank of *any* changed file in the commit that is related to a bug fix. To investigate whether this approach is justified, we studied SWT bug reports used by previous researchers and found that about 40% of the bugs have more than one changed file. We inspected bugs with at least one changed file ranked at the top of the ranked list and found that 11 out of 33 bugs have multiple changed files and, for each of them, only 1 file indeed contains the causes of the failure described in the bug report. This result is consistent with a previous study [9], where 28% of the files changed for a bug fix where not directly related to the actual bug fix.

Table 2: SWT bug report categories.

	Stack Trace	Program Entity	Test Case	Natural Language
Top 1	0 (0%)	35 (35.7%)	7 (7.1%)	1 (1.0%)
Top 5	0 (0%)	62 (63.3%)	9 (9.2%)	7 (7.1%)
Top 10	3 (3.1%)	71 (72.4%)	10 (10.2%)	8 (8.2%)
Total	4 (4.1%)	83 (84.7%)	10 (10.2%)	14 (14.3%)

Table 3: ZXing bug report categories.

	Stack Trace	Program Entity	Test Case	Natural Language
Top 1	0 (0%)	3 (15%)	1 (5%)	0 (0%)
Top 5	0 (0%)	7 (35%)	2 (10%)	2 (10%)
Top 10	0 (0%)	8 (40%)	2 (10%)	5 (25%)
Total	0 (0%)	9 (45%)	3 (15%)	9 (45%)

Table 4: JodaTime bug report categories.

	Stack Trace	Program Entity	Test Case	Natural Language
Top 1	0 (0%)	2 (22.2%)	2 (22.2%)	0 (0%)
Top 5	0 (0%)	5 (55.6%)	3 (%)	0 (0%)
Top 10	0 (0%)	7 (77.8%)	4 (44.5%)	0 (0%)
Total	0 (0%)	8 (88.9%)	4 (44.5%)	1 (11.1%)

Table 5: AspectJ bug report categories.

	Stack Trace	Program Entity	Test Case	Natural Language
Top 1	7 (2.4%)	38 (13.3%)	17 (5.9%)	9 (3.1%)
Top 5	14 (4.9%)	56 (19.6%)	25 (8.7%)	22 (7.7%)
Top 10	18 (6.3%)	61 (21.3%)	30 (10.5%)	27 (9.5%)
Total	78 (27.3%)	145 (50.9%)	138 (48.3%)	66 (23.1%)

In general, when multiple files are changed, the changed files are not necessarily all faulty files and, in particular, the best-ranked file may not be faulty at all. This is an important factor to be considered when evaluating IR-based fault localization techniques. Although there is some initial evidence that this may not always have a significant effect on the fault localization results [9], considering unrelated files is less than ideal, and can skew the result in at least some cases.

4.2 Data Categorization and Ranking

Intuitively, whether an IR-based approach can work effectively in fault localization will heavily depend on whether the bug reports are good enough to be used as query keywords. To investigate whether this intuition is correct, we examined bug reports from the benchmarks described above and categorized the information contained in them, which includes stack traces, program entity names, test cases, and pure natural language descriptions. We call stack traces, program entity names, and test cases *identifiable information* because these three types of information can be directly matched to classes, methods, or variables in the program.

We associated the information contained in the bug reports with the results generated by BugLocator [25] and tried to find whether this information affected the results.[1] We selected those bugs whose faulty files were ranked in the Top 1, Top 5, and Top 10 and examined whether their reports contained the information mentioned above. The reason we used absolute rank instead of percentage rank is that the programs we studied are relatively large, so even Top 1% consists of at least 20 files. As we observed later in our user study (see Section 5.5), people rarely looked past the first 3 suggestions. Therefore, absolute rank makes more sense in terms of helping developers to narrow down the search space.

The results are presented in Tables 2 through 5, where each table describes the bug reports for a benchmark program. For each program, the corresponding table shows how many bug reports match each information category (columns), divided by fault localization performance (rows). For example, for SWT, there was only one bug report that contained natural language and achieved a perfect top rank, whereas 35 bugs that contained program elements achieved a perfect top rank. In the tables, the total does not sum up to 100% because there is overlap among bug reports containing stack traces, test cases, and program entity names. All percentages are computed over the total number of bug reports.

[1]We used the exact results generated by BugLocator and did not filter out the changed files that are not buggy because this filtering was not part of the technique(s) considered.

4.3 Analysis

After we manually categorized our bug reports based on the information they contained, we performed a data analysis to evaluate Hypothesis 1 and answer RQ1.

Bug report category determines localization result.

If Hypothesis 1 were true, we would expect to see that IR-based fault localization generated better ranked lists when the input bug reports contained the rich information we mentioned above. Both SWT and AspectJ contain a considerable numbers of bugs, so we performed an unpaired t-test on the two projects in order to assess whether the presence of richer information caused significant differences in the results of IR-based fault localization.

The average ranking of faulty files was 6.64 and 21.85 for bugs in SWT whose reports contain and do not contain the program entity names, respectively. The difference is statistically significant by a t-test ($p < 0.05$). Similarly, for bug reports containing the same type of information in AspectJ, the average ranking was 43.10 and 156.42, respectively, which is also statistically significant ($p < 0.05$).

However, the difference caused by the presence of stack traces and test cases was not statistically significant, despite the fact that the average ranking of faulty files is indeed better when such information is present. We also compared the difference caused by combinations of different types of information. Interestingly, after we removed the bug reports containing stack traces and test cases, the average rankings became 26.65 and 120.44, and the difference between pure program entity names and natural language became highly statistically significant ($p < 0.01$).

The reason behind this, as we observed, is that some stack traces and test cases contain many class names and method names, and only a small subset of the names are closely related to the bug. The other names created noise for IR-based techniques when they tried to compute the textual similarity. Therefore, stack traces and test cases do not always help IR-based techniques even though they contain very rich information.

To conclude, we found *some support for Hypothesis 1* according to the above analysis. The effectiveness of IR-based techniques is affected by the quality of bug reports, but not all good bug reports can produce good fault localization results.

Program entity names help generate good fault localization results.

Although in the context of Hypothesis 1 we found that stack traces and test cases may not always help improve IR-based fault localization results, there are some cases where they do generate good ranked lists. We use SWT and AspectJ as examples to illustrate how the information contained in bug reports may affect fault localization results.

For SWT, among the bugs whose faulty files are ranked Top 1 by the IR-based approach, 35 of the associated reports contain program entity names, such as classes or methods, most of which are indeed faulty. In 7 bug reports, the users reported the bug along with test cases to reproduce the bug. For all 7 bugs, the classes containing the bugs are explicitly used in the provided test cases.

For AspectJ, 7 out of 49 bug reports for which the IR-based approach ranked the faulty files as Top 1 contain stack traces. After comparing the files on the stack and the actual faulty files, we noticed that all the 7 faulty files appear more frequently on the stack than any other file. For the bug reports that ranked the faulty files in the Top 10, the names of the faulty files also occur many times, and the IR-based technique was therefore able to rank them high.

For almost all the benchmark programs, very few bug reports that contain only pure natural language descriptions produced good fault localization results. On average, bug reports containing only natural language account for 11% of Top 1 results, 18% of Top 5 results, and 20% of Top 10 results. All other bug reports producing good fault localization results contain at least one piece of information of type stack traces, test cases, or program entity names.

Therefore, the answer to RQ1 is that *program entity names in a bug report tend to result in good ranked lists when used for IR-based fault localization; stack traces and test cases are useful information, but they do not always help.*

If bug reports that produce good fault localization results contain identifiable information, such as program entity names, and sometimes stack traces or user provided test cases, we would like to assess how often such information is present in bug reports. According to a study by Saha and colleagues [20], more than 80% of bug reports contain an exact match for at least one of the files to be fixed. However, their study was performed on the dataset selected by previous researchers using certain criteria, instead of on bugs randomly chosen from bug repositories. Therefore, their results might not apply for other real-world bug reports.

To investigate further, we extracted 10,000 SWT bug reports from Bugzilla (`https://www.eclipse.org/swt/bugs.php`) and gathered statistics about the information contained in these bug reports. Our data shows that 10% of the bug reports contain stack traces, 3% contain test cases, and 45% contain the names of program entities (including those containing stack traces and test cases). The remaining 55% of bug reports consist of pure natural language.

With a large portion of bug reports not containing enough identifiable information, our data do not support those previously presented results. According to such data, IR-based approaches are unlikely to be effective in a majority of cases.

5. USER STUDY

To answer RQ2 and RQ3, we performed a user study. In this section, we discuss the study and our findings. We first present the program that we used in the study, our participants, the study setting, and our evaluation metrics. We then discuss our results in detail.

5.1 Benchmark Program

We chose SWT as our benchmark program because it is a well maintained open source project with an associated bug tracking system. At the same time, as a graphical widget toolkit, SWT is neither too trivial nor too complicated for users who are not familiar with it.

5.2 Participants

The participants in our study were students enrolled in graduate-level software engineering classes. At the time of the study, they were already familiar with software debugging and had learned the concept of automated fault localization. We recruited students from both regular on campus classes and Georgia Tech's OMSCS program (`http://www.omscs.gatech.edu/`). It is important to note that most students enrolled in the OMSCS program are professional developers with many years of software development experience. Overall, we had 70 participants in total with various programming and working experience. In the rest of this section, we will use "participants" and "users" interchangeably to refer to the participants.

5.3 Study Settings

5.3.1 Variables and Bug Selection

Our experimental study includes both experiment-related variables and bug-related variables, as follows.

The ranked list that is generated by the fault localization technique: the user is either given the list (Y) or not given the list (N). To the best of our knowledge, most IR-based techniques present their results in the form of a ranked list. Therefore we use a ranked list to represent the results of the technique.

The information that a bug report contains: only natural language descriptions (NL) or program entity names (PE). Based on the answer to RQ1, we know that IR-based techniques perform better with the names of program entities, so we wanted to see whether the users would also perform better with the same information.

The fault localization result that the fault localization technique produces: a good result (the faulty file ranks first, G) or a bad result (the faulty file ranks below the Top 10, B).

By combining the above two variables, we identified four types of bug reports to be selected:

- PE-G: containing the program entity names and producing good fault localization results.
- PE-B: containing the program entity names and producing bad fault localization results.

Table 6: User study groups.

Group	Task 1	Task 2
1	PE-G-Y-1	PE-G-N-2
2	PE-B-Y-3	PE-B-N-4
3	NL-G-Y-5	NL-G-N-6
4	NL-B-Y-7	NL-B-N-8
5	PE-G-Y-2	PE-G-N-1
6	PE-B-Y-4	PE-B-N-3
7	NL-G-Y-6	NL-G-N-5
8	NL-B-Y-8	NL-B-N-7

- NL-G: containing only natural language descriptions and producing good fault-localization results.
- NL-B: containing only natural language descriptions and producing bad fault-localization results.

We selected 8 bugs (bug IDs 1–8), 2 for each type, based on the following criteria:

- We were able to find the code revision directly fixing the bug. We searched the code repository of SWT with the bug ID, assuming that the developers would mention the ID in the comments after they fixed a certain bug.
- The bugs involved only one buggy method in one file, to make the tasks simpler.
- The buggy code revision worked, and we were able to reproduce the bug on the Linux platform that we used in the virtual machine provided to the study participants.

5.3.2 Experimental Groups

The participants we recruited were assigned to 8 groups that we designed based on the variables we identified, which are shown in Table 6.

For each group, the users were asked to finish two tasks, one with the help of a ranked list and the other without. The bugs in the two tasks were of the same type in terms of the variable combination, to mitigate the bias caused by the bug itself.

5.3.3 Data Collection

Survey Form: We used a survey form for participants to submit their debugging results. The participants were asked to provide the name of the file and method, and the line number(s) as well as their confidence about the results they provided. We assess the results in Section 5.5.

Eclipse Plugin: We created an Eclipse plugin called DebugRecorder to log the participants' debugging activities. DebugRecorder can record mouse and keyboard events produced by the participants, along with the elements associated with these events. By analyzing the events, we can get a list of files that the users viewed, the locations in the files on which they clicked, and their keyboard input as they were fixing the bugs. Besides recording the debugging activities, DebugRecorder also visualizes within Eclipse the list of suspicious files computed by BugLocator; by clicking on a file in the list, the user can navigate directly to the file.

5.3.4 Procedure

To avoid the need for the study participants to configure the environment, we created a virtual machine in which we set up the projects, installed Eclipse, and added to Eclipse the DebugRecorder plugin. A user was created for each of the eight groups to access the projects assigned to them with pre-assigned credentials. Once logged in, the participants could follow the detailed instructions and start their debugging tasks. For each debugging task, the participants were given a bug report and a test case reproducing the bug. For one of the two tasks, they were given a ranked list based on which group they were assigned to. The debugging time for each task was recommended to be, but not restricted to, 30 minutes. After the users finished the tasks, the plugin automatically sent the recorded data to a server at Georgia Tech, and the participants were guided to provide their debugging results using the survey form. Finally, the users were asked to give feedback about their experience using the tool. The recorded logs and survey responses were associated with a randomly generated ID to keep all data anonymous.

5.4 Evaluation Metrics

To assist answering RQ2 and RQ3, we defined a set of metrics and used them to analyze the data we collected from the study. We introduce these metrics in the rest of this section.

5.4.1 Score of Debugging Results

To evaluate how well the developers performed in the study, we used the following equation:

$$score = 0.3 \times class + 0.3 \times method + 0.4 \times (1 - \frac{distance}{1000}) \quad (1)$$

For each debugging result, we compared the file and method names with our candidate bug location, identified according to the fix committed by SWT's actual developers. The users received 0.3 points if the file name they provided was the same as the name of the candidate file. Another 0.3 would be given to the users if they gave the same method name as the candidate fix. Finally, the users received up to 0.4 points based on how close the location they provided was to the candidate bug location; we measured distance as the number of statements along the shortest path between the two locations in the interprocedural control flow graph. We assume that two locations are not closely related if they are more than 1000 statements away from each other.[2]

For the cases where the users found a different location, we did not consider the location completely correct even if they fixed the bug. We assumed that the fix committed to the repository was the best since it was the one performed by the actual developers.

5.4.2 List of Files Viewed

To investigate whether the participants navigated the files in a different way when they were given (or not given) the ranked list, we analyzed the list of source files that the users viewed while they were performing their debugging tasks, which is contained in the logs recorded by our Eclipse plugin.

5.4.3 Time Used to Debug

When comparing the users' debugging performance, we also want to know the time it took them to find the bug, in addition to knowing their debugging results.

We divided the users' debugging activities into two steps: finding the right file and locating the bug. We calculated the

[2]We experimented with different values and found that this parameter did not significantly affect the results.

Table 7: Scores for debugging results.

	PE-G	PE-B	NL-G	NL-B	Overall
w/ list	0.47	0.20	0.22	0.09	0.24
w/o list	0.46	0.29	0.15	0.17	0.27

Table 8: Time used to debug (in minutes).

	File Focused		Bug Found	
	w/ list	w/o list	w/ list	w/o list
	min/med/avg	min/med/avg	min/med/avg	min/med/avg
PE-G	1/1/4	1/4/6	17/67/76	10/35/63
PE-B	1/15/51	5/17/30	23/97/106	13/35/89
NL-G	1/9/9	3/20/15	18/120/87	30/120/87
NL-B	23/64/68	1/120/93	13/120/110	22/120/103

time used in each step separately and compared the performance in terms of time used by the participants with the list and those without it.

Time necessary to find and focus on the faulty file: We calculated the time elapsed from the moment in which a user started a task to the moment in which he or she started focusing on the actual faulty file. By focusing on a file we mean that the user viewed the file continuously for more than one minute. If the user did not focus on the faulty file at all, the time was set to 2 hours.

Time necessary to find the bug: We calculated the time elapsed from the moment in which a user started focusing on the faulty file to the moment in which he or she located the bug. If the user did not find the bug, we set the duration to 2 hours.

5.5 Results

We ran our experiment with 70 students, divided into 8 groups for evaluating Hypotheses 2 and 3, and the corresponding Research Questions 2 and 3; there were 9 students in Groups 1–6, and 8 students in Groups 7 and 8. We examined their debugging results and found some invalid entries. Specifically, 2 participants misunderstood our instructions and changed the test cases to make them pass, 2 reported that they could not reproduce the bugs, and 8 submitted incomplete data (just finished one task or did not upload the plugin data to our server). We removed the invalid data and ended up having valid data from 58 participants, whose distribution in Groups 1-8 is 8, 7, 8, 8, 6, 8, 6 and 7, respectively. We analyzed the valid results using the metrics defined in Section 5.4. An overview of the results is presented in Tables 7 and 8.

Table 7 shows the average scores of the debugging results for users working on different types of bugs with and without the help of the ranked lists generated by BugLocator. The minimum, median, and average time consumed by the users in the two steps mentioned in Section 5.4 is shown in Table 8. We do not list the maximum time because, in each group, there was at least one participant who did not find the bug, so the maximum time is always 120 minutes.

5.5.1 Good Bug Reports Improved Users' Debugging Performance

If Hypothesis 2 were correct, developers should perform better on tasks when aided by bug reports that contained more useful information. To evaluate this hypothesis, we compared the time to find the bug, the average score of the answer, and the time to focus on the faulty file (see Section 5.4) between groups where no list was provided. Therefore, we were measuring how much the quality of the bug report influenced the ability of the developer to find the bug.

Without the list, the average time used to find the bug for PE-G and NL-G showed no statistically significant difference. However, the average score was 0.46 for groups with PE-G bug reports (1, 5), 0.15 for those with NL-G bug reports (3, 7), and the difference was statistically significant based on an unpaired t-test ($p < 0.05$). In addition, the average time necessary to focus on the faulty file was 5.55 minutes for PE-G, 14 minutes for NL-G, and the difference was also in this case statistically significant ($p < 0.05$).

The comparison between PE-B groups (2, 6) and NL-B groups (4, 8) shows that there is a statistically significant difference ($p < 0.05$) in the time needed to focus on the faulty file. However, the difference in the score of the debugging results and time used to find the bug was not statistically significant.

Based on these results, we found *support for Hypothesis 2* in the difference in time necessary to focus on the faulty file. However, the presence of program entity names only helps to shorten the time to locate the file.

5.5.2 Good Ranked Lists Helped When Users Could Not Get Enough Hints From Bug Reports

To test Hypothesis 3, we wanted to show that participants performed better when they debugged with the help of a ranked list generated by an IR-based technique. Comparing the performance of groups debugging the same task with and without the aid of a ranked list (1 vs 5, 2 vs 6, 3 vs 7, and 4 vs 8) helped us to study this hypothesis.

For the average score of the debugging results, as well as the time used to locate the bug, the comparison of all pairs showed no statistically significant difference by an unpaired t-test. The only exceptions are the NL-G groups (3 and 7). The average time needed by the users debugging NL-G tasks with the help of the list was 9 minutes, whereas the time necessary to participants without the list was 15 minutes (statistically significant with $p < 0.05$).

Summarizing the results, the ranked list helped developers only under certain circumstances. Therefore, we found *support for Hypothesis 3 only when the bug report does not contain rich information*, such as the program entity names, and when the list contains the faulty file with Rank 1. In all other cases considered, we found *no support* for this hypothesis.

5.5.3 Users Took Advantage of the Presence of Program Entity Names by Using Them as Search Keywords.

The evaluation of Hypothesis 2 already showed that rich information influenced developers' debugging performance. Next, we want to answer RQ2 by analyzing how performance was affected. In the following, we only consider the data corresponding to tasks performed without the aid of ranked lists. We examined the logs from our plugin and the surveys to understand performance.

An analysis of the log file shows that the participants used the name of the program entity that appears in the bug report as a keyword for searching source files. They quickly focused on the file with an exactly matching name. If the name appearing in the report was not a file name, they searched files containing the entity name and took longer to focus on the right file. The participants using a bug report with only natural language, conversely, clicked on many files at the beginning before they started focusing on any file. Even after they focused on the faulty file, they still moved to other files and went back and forth many times.

Another observation is that, when the IR-based technique did not rank the faulty file first, the participants also did poorly, no matter whether or not they had the help of the generated list. This happened because the bug reports contained misleading information, and IR could not retrieve the correct file using such information. The users were also guided to the wrong file by the information in the bug reports.

In summary, the presence of program entity names in the bug reports did improve the developers' debugging performance by giving them explicit guidance on where and how they should start looking and confidence that they were looking in the right place.

5.5.4 Perfect Ranked Lists Improve Performance in Limited Cases

To answer RQ3, which is about the influence of the ranked list on users' debugging behavior, we did the same pairs of comparisons as for Hypothesis 3 and analyzed the data in more detail. Specifically, we analyzed how the debugging time, accuracy, and behavior were influenced by the presence or absence of the ranked lists.

According to the results in Table 8, with the help of the list, the time taken for the participants to focus on the faulty file was 2 minutes shorter when the bug report itself contained the program entity names, and 6 minutes shorter when the bug report only contained natural language descriptions. Both cases show that the ranked list helped users to focus on the faulty files quickly. For most of the cases with high-ranked bug reports, we see that there is almost no difference between debugging accuracy when using a ranked lists or not using it (average 0.24 and 0.27). However, the presence of the ranked list actually lengthened the time necessary to find the bug. Moreover, for cases in which the IR-based technique did not rank the faulty file in the Top 10, the participants performed worse then using the list.

This is easy to understand for cases in which the ranked list was inaccurate, as the participants tended to focus on the wrong files even after they viewed the faulty file. If the list was accurate, however, the participants easily identified the right file, but they spent more time locating the bug in the file. The time to locate the bug for NL-G groups was the same with or without the list because the confidence given by the list avoided wasting time on other files (see Section 6.1), which compensated for the extra time used to understand the bug. In general, considering the time spent in the two steps identified in Section 5.4 makes the results of existing IR-based fault localization techniques sound less promising, as they help developers perform only the least expensive of these two steps (finding the right file).

After examining the lists of files that the participants viewed while they were debugging, we noticed that there was a difference for users with and without the help of the ranked list. This difference indicates that the list did affect the users' debugging behaviors by leading them to some files they would otherwise not consider. The difference is, however, rather small. For the same bug, most of the files viewed were the same in the two cases. A further analysis of the log files shows that the difference exists mainly because the users with the ranked list would click on the files ranked at the top of the list. After that, they would switch to the same file(s) viewed by the users without the lists

To summarize, the ranked lists generated by the IR based fault localization techniques do not always help users debug; they only help when the techniques can generate perfect lists for bugs without rich, identifiable information in the reports. Moreover, if the generated list is not good enough, it can even harm developers' performance by leading them to focus on the wrong files.

6. DISCUSSION

6.1 Implications

The quality of bug reports matters.

As shown in our analytical and user study, the contents of the bug reports have a strong influence on both the effectiveness of automated techniques and the performance of the developers. On the one hand, if a bug report contains identifiable information, such as program entity names, IR-based techniques tend to generate good fault localization results by directly matching the name to source files with or containing the same name. On the other hand, developers can also perform better with a bug report with such information by directly taking advantage of emergent hints in the report.

Conversely, both developers and automated techniques suffer in the presence of poor quality bug reports. Poorly written reports contain much irrelevant and misleading information, which unintentionally misleads both IR and developers.

Therefore, *developing approaches that help users write good bug reports seems to be more urgent than further investigating IR-based techniques.*

Developers are smart enough to locate faulty files quickly when given a rich bug report.

When a bug report contains rich information, such as names of program entities, it is often obvious to the developers where they should start looking. Many bug reports that can generate perfect ranked lists already contain the names of the faulty files or methods. It is very likely that the developers would look at these files and entities mentioned in the bug reports first anyway, no matter whether they have additional help from an automated technique. If the developers happen to be those who developed the project, and are thus very familiar with it, they should be able to find the relevant file(s) immediately. Even if the developers are not familiar with the project, they can probably still find the relevant file(s) quickly using keyword search, which was the case in our user study.

If the bug reports contain names of more than one program entity, the developers can sometimes do a better job than the IR-based techniques. Despite their latest improvements, IR-based techniques still use textual similarity between bug reports and source code to rank files. When multiple program entity names exist, such as in a bug report containing stack traces or test cases, pure textual similarity may not be able to distinguish the actual buggy file from other files that are similar but unrelated. Humans, on the other hand, can often tell which name in the report is more suspicious based on the structure of the sentences and their understanding of the reports.

IR-based fault localization should focus more on improving results for bug reports without identifiable information.

The users did much worse when they worked on the tasks with bug reports containing little information that can be used to directly identify related program entities. The lack of hints in the bug reports has two main effects on the developers' performance when debugging. First, developers spend much more time finding the file containing the bug. Second, even when they find the buggy file, they are unsure about their own decisions. They go back and forth among different files, either to find files which might seem more suspicious, or to gain confidence in the file they found. In the logs recorded by the plugin, users with less information focused on many more files, both before or after they got to the real faulty file. They ended up spending more time in both steps than the users with identifiable information in the bug reports. In the post-task survey, several users without the help of the list mentioned that they kept investigating other files because they did not know whether what they found was correct or not.

The ranked list helped improve the confidence of the users debugging with less information. Although the average time used to find the bug was in the end almost the same, we found that users with the help of perfect ranked list spent more time on the faulty file, which is also the top ranked file in the given list. Users given a bad ranked list also focused on the top file much longer than other files, including the faulty file.

When the bug report at hand does not contain obvious bug information, fault localization gives developers both a starting point and the confidence to focus on that point. Therefore *IR-based fault localization should focus on providing better results for bug reports without identifiable information, so as to give developers a good starting point and avoid giving them a false sense of confidence.*

The participants had a hard time using the ranked list generated by the IR-based technique.

There were several issues that limited the usefulness of the ranked list. First, the lack of context information did not help users' understanding of the bug. An interesting finding from the recorded data is that some users clicked the first file on the list and quickly switched to other files, no matter whether the file was actually faulty or not. This happened when the bug reports contained only natural language descriptions. These users would come back to the files and focus on them some time later, but they also focused on other files before that. According to the responses of the survey form from these users, when presented with a (possibly large) source file, they felt overwhelmed and did not know where to start inspecting the code. Only after they had focused on some other files executed before the top-ranked files and had an understanding of the context, they would focus on the files in the list.

Second, the representation affected the way the ranked list was used by the participants. The feedback from the participants showed that the suspicious score for the top ranked files influenced the amount of time they spent focusing on the files. Specifically, if the suspiciousness of a file was 1, the users tended to spend considerably more time on the file than they would have spent on a file with lower suspiciousness. Besides, some users said they gave up using the list after they looked at the 2 or 3 files at the top of the list but were unable to figure out where the bug was. In fact, an examination of the log files showed that all participants clicked on at most 3 files in the list.

In summary, *developers tend to give up quickly on IR-based techniques. If these techniques want to be successful, they have to provide not only accurate ranked lists of files, but also finer-grained information and context to help developers localize faults quickly.*

6.2 Threats To Validity

There are several potential threats to the validity of our study.

Different companies may provide different formats or forms for their users to fill in when they submit a bug report. We only examined extensively the bug reports in one project, so our findings on bug report information may not generalize to other projects.

We only analyzed the results generated by one IR-based tool, BugLocator; the results may not generalize to other IR-based approaches. However, many other IR-based techniques are developed on the same assumptions that users can provide rich information in their bug reports and are evaluated on the same dataset used by BugLocator. We believe that BugLocator is a good representative of IR-based fault localization techniques in general.

Although we also had several professional developers among the participants in our user study, most participants were graduate students. The involvement of OMSCS students helped us mitigate this threat, as most OMSCS students have years of professional development experience.

Our participants were not familiar with the projects they debugged, which may not be the case in real-world situations. It is however not completely uncommon for developers to debug someone else's code.

When there were multiple ways to fix a bug, we assumed that the one we extracted from the project repository was the best one, as it was the actual developers' fix. This may not always be the case, and some of our participants might have found a better fix, which we however did not consider as completely correct.

7. RELATED WORK

7.1 IR-Based Fault Localization Techniques

In recent years, researchers have been working on applying information retrieval techniques to fault localization. Different IR models such as the Vector Space Model (VSM) [16], Latent Semantic Indexing (LSI) [2, 16] and Latent Dirichlet Allocation (LDA) [13] have been proposed and applied by researchers to successfully retrieving faulty files.

Rao and Kak [18] compared 5 different text models in terms of fault localization on a large number of real-world bugs from the iBUGS dataset [3]. Their comparison shows that IR-based bug localization techniques are at least as effective as other fault localization approaches. They also found that complicated models do not perform better than simple models such as Unigram or VSM.

Zhou and colleagues [25] revised the original VSM model by taking into account the sizes of source files and integrating the similarity between a new bug and previously fixed bugs. Their revisions are based on two observations. The first observation is that source files with larger size are more likely to contain bugs. Their second observation is that previously fixed bugs could help locate the relevant files for a new bug in cases in which the new bug is similar to the fixed ones. They developed BugLocator, a tool implementing their technique, and evaluated the tool on four open source projects. Their results showed that BugLocator outperformed previous IR-based fault localization techniques.

Saha and colleagues [20] further improved the results of Zhou and colleagues by incorporating structural information from the source code. The main insight of their improvement is that source files are structured documents, and code constructs such as class and method names can help improve the accuracy of fault localization. The evaluation on the same benchmarks used by BugLocator showed the effectiveness of their technique.

In a follow-up study, Saha and colleagues [19] investigated whether the IR-based techniques also work for programs written in languages that are not object-oriented. They created a dataset containing 7500 bugs from five popular C projects and evaluated their previous work on the dataset. While IR-based fault localization was still effective, the integration of program structure information did not help as much in C program as in Java programs.

7.2 User Studies on Fault Localization Techniques and Tools

Despite the fact that researchers are devoting a significant amount of effort to improving automated fault localization techniques, only a few groups conducted experimental studies to evaluate their fault localization techniques and tools with developers [5, 10, 22, 23]. The studies either involved a small number of participants (less than 10) or were conducted on trivial programs with less than 100 lines of code.

A relatively extensive user study was conducted to evaluate Whyline [8], an interactive debugging tool allowing users to ask questions about program behaviors. In the study, 20 participants were asked to investigate two real bugs in a large project: ArgoUML (150 kLOC). The study results showed that participants that used Whyline performed better than those without the help of tool.

A previous study of two of the authors [17] assessed the practical usefulness of a family of spectra-based debugging techniques through a user study. The study showed that spectra-based techniques may be based on assumptions, such as "perfect bug understanding," that are too strong to hold in practice. It also showed that developers want explanations, rather than just a list of ranked suspicious statements.

Gouveia and colleagues [6] proposed a visualization for displaying the reports generated by existing fault localization techniques. They used three types of visualizations—sunburst, vertical partition, and bubble hierarchy—to display the program structure and show the fault localization results on top of that, making it easier for developers to understand the organization of the code and locate a suspicious program entity more quickly. They also performed a user study with 40 students, who had to locate an injected one-line bug in a 17 kLOC program. Their results showed that the students using the proposed visualizations debugged faster and better than those who used only traditional debuggers.

Although researchers are aware of the important role that developers play in the debugging process, evaluations involving developers are still fairly rare. This is especially true for IR-based techniques, which makes it difficult to know whether these techniques could work in practice. In this paper, we took a first step in this direction, and tried to answer some of the open questions about IR-based fault localization techniques.

8. CONCLUSIONS

Fault localization techniques based on information retrieval have been attracting increasing attention in the last few years, and several techniques have been presented in this arena. As for any new approach, the initial evaluation of these techniques has been somehow limited and performed only analytically. There are, therefore, several questions about these techniques that have not yet been answered. What type of bug reports are amenable to IR-based fault localization? Are the bug reports that work well with these techniques good enough to be directly used by developers? Can developers really take advantage of the information provided by these techniques to debug more effectively?

To answer these questions, this paper presents two in-depth studies: an analytical study and a user study. The results of our analytical study show that the information needed for IR-based techniques to be effective is often not available in bug reports, which may limit the applicability of these techniques. Even when these high-quality bug reports are available, and IR-based techniques can generate "perfect" fault-localization results, they may still benefit developers only marginally. The reason for this is that high-quality bug reports are often good enough to guide developers to the file in which the bug is located without any additional help. In addition, the list of suspicious files produced by the IR-based technique may help developers get to the right file faster, but does not help them speeding up the localization of the bug within that file, which we have found to be the most time consuming part of debugging.

One of our findings, in this work, is that trying to define a one-size-fits-all, ideal automated-debugging technique is unlikely to be a worthy pursuit. We believe that a successful debugging approach must be able to account for the variability in the artifacts available to developers and their quality (*e.g.,* in terms of test cases and bug reports). Researchers should also consider other ways in which developers might want to formulate queries beyond implicit ones, such as bug reports, and propose more interactive debugging approaches. Based on our results, and on findings from related previous work [17], we also conclude that it is of fundamental importance to evaluate debugging techniques through user studies. Without such studies, it is simply not possible to assess the actual usefulness of an approach. In addition, user studies allow researchers to identify actual needs that they might not have realized, and whose realization can lead to new research venues and, ultimately, to the development of more effective debugging techniques.

Acknowledgements

This work was partially supported by NSF awards CCF-1320783 and CCF-1161821, and by funding from Google, IBM Research, and Microsoft Research to Georgia Tech.

9. REFERENCES

[1] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3:993–1022, Mar. 2003.

[2] B. Cleary, C. Exton, J. Buckley, and M. English. An empirical analysis of information retrieval based concept location techniques in software comprehension. *Empirical Softw. Engg.*, 14(1):93–130, Feb. 2009.

[3] V. Dallmeier and T. Zimmermann. Extraction of bug localization benchmarks from history. In *Proceedings of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, pages 433–436, New York, NY, USA, 2007. ACM.

[4] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman. Indexing by latent semantic analysis. *JOURNAL OF THE AMERICAN SOCIETY FOR INFORMATION SCIENCE*, 41(6):391–407, 1990.

[5] M. A. Francel and S. Rugaber. The value of slicing while debugging. *Sci. Comput. Program.*, 40(2-3):151–169, July 2001.

[6] C. Gouveia, J. Campos, and R. Abreu. Using html5 visualizations in software fault localization. In *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on*, pages 1–10, Sept 2013.

[7] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, ASE '05, pages 273–282, New York, NY, USA, 2005. ACM.

[8] A. J. Ko and B. A. Myers. Designing the whyline: A debugging interface for asking questions about program behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '04, pages 151–158, New York, NY, USA, 2004. ACM.

[9] P. S. Kochhar, Y. Tian, and D. Lo. Potential biases in bug localization: Do they matter? In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, pages 803–814, 2014.

[10] S. Kusumoto, A. Nishimatsu, K. Nishie, and K. Inoue. Experimental evaluation of program slicing for fault localization, 2002.

[11] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan. Scalable statistical bug isolation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 15–26, New York, NY, USA, 2005. ACM.

[12] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff. Sober: Statistical model-based bug localization. *SIGSOFT Softw. Eng. Notes*, 30(5):286–295, Sept. 2005.

[13] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Source code retrieval for bug localization using latent dirichlet allocation. In *Proceedings of the 2008 15th Working Conference on Reverse Engineering*, WCRE '08, pages 155–164, Washington, DC, USA, 2008. IEEE Computer Society.

[14] C. D. Manning, P. Raghavan, and H. Schütze. *Introduction to Information Retrieval.* Cambridge University Press, New York, NY, USA, 2008.

[15] C. D. Manning and H. Schütze. *Foundations of Statistical Natural Language Processing.* The MIT Press, Cambridge, Massachusetts, 1999.

[16] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic. An information retrieval approach to concept location in source code. In *Proceedings of the 11th Working Conference on Reverse Engineering*, WCRE '04, pages 214–223, Washington, DC, USA, 2004. IEEE Computer Society.

[17] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 199–209, New York, NY, USA, 2011. ACM.

[18] S. Rao and A. Kak. Retrieval from software libraries for bug localization: A comparative study of generic and composite text models. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 43–52, New York, NY, USA, 2011. ACM.

[19] R. Saha, J. L. Lawall, S. Khurshid, and D. E. Perry. On the Effectiveness of Information Retrieval Based Bug Localization for C Programs. In *ICSME 2014 - 30th International Conference on Software Maintenance and Evolution*, pages 161–170, Victoria, Canada, Sept. 2014. IEEE.

[20] R. Saha, M. Lease, S. Khurshid, and D. Perry. Improving bug localization using structured information retrieval. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 345–355, Nov 2013.

[21] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Commun. ACM*, 18(11):613–620, Nov. 1975.

[22] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.

[23] M. Weiser and J. Lyle. Experiments on slicing-based debugging aids. In *Papers Presented at the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers*, pages 187–197, Norwood, NJ, USA, 1986. Ablex Publishing Corp.

[24] 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. Technical Report RN/14/14, University College London, 2014.

[25] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 14–24, Piscataway, NJ, USA, 2012. IEEE Press.