2012 28th IEEE International Conference on Software Maintenance (ICSM)

Interactive Fault Localization Leveraging Simple User Feedback

Liang Gong[1], David Lo[2], Lingxiao Jiang[2], Hongyu Zhang[1]
[1]School of Software, Tsinghua University, Beijing 100084, China
Tsinghua National Laboratory for Information Science and Technology (TNList)
[2]School of Information Systems, Singapore Management University, Singapore
{gongliang10@mails,hongyu@}tsinghua.edu.cn,{davidlo,lxjiang}@smu.edu.sg

***Abstract*—Many fault localization methods have been proposed in the literature. These methods take in a set of program execution profiles and output a list of suspicious program elements. The list of program elements ranked by their suspiciousness is then presented to developers for manual inspection. Currently, the suspicious elements are ranked in a batch process where developers' inspection efforts are rarely utilized for ranking. The inaccuracy and *static* nature of existing fault localization methods prompt us to incorporate user feedback to improve the accuracy of the existing methods.**

In this paper, we propose an *interactive* fault localization framework that leverages simple user feedback. Our framework only needs users to label the statements examined as faulty or clean, which does not require additional effort than conventional non-interactive methods. After users label suspicious program elements as faulty or clean, our framework incorporates such information and re-orders the rest of the suspicious program elements, aiming to expose truly faulty elements earlier. We have integrated our solution with three well-known fault localization methods: *Ochiai*, *Tarantula*, and *Jaccard*. The evaluation on five Unix programs and the Siemens test suite shows that our solution achieves significant improvements on fault localization accuracy.

I. Introduction

There has been a growing concern about the impact of rapid increase in the complexity of software, which leads to difficulties in writing correct computer programs. To improve the quality of software systems, software testing, debugging, and verification are widely used. However, these activities are often labor-intensive, accounting for 30% to 90% of labor spent for a project [6], [7].

When a software error is revealed by testing or reported by an end-user, the very first step of diagnosis is to locate the root cause. However, this step also requires much manual effort [10]. To address this issue, automatic fault localization tools, which aim to reduce corrective maintenance cost and create robust software more efficiently, have been proposed.

There are many fault localization techniques in the literature. *Spectrum-based Fault Localization* (abbr. SBFL *e.g.*, [2], [4], [5]) is one kind of automated debugging techniques that effectively narrows down the possible locations of software faults and thus helps save developers' time. This type of techniques usually takes in a set of program execution profiles along with their labels (*i.e.*, *pass*ed or *fail*ed executions), and recommends suspicious program elements to developers for manual inspection.

Although many spectrum-based fault localization techniques have been reported to be helpful [1], [13], [15], [16], there are still many drawbacks hindering their massive application in industry. One of the most important issues is the unsatisfactory accuracy of existing statistical fault localization techniques [26]. This can partially be attributed to the limited information provided by program spectra as debugging often requires comprehensive understanding of program semantics. Unfortunately, artificial intelligence and machine learning techniques employed for fault localization are still unable to automatically and completely comprehend program semantics and its intended behavior.

Despite the fact that spectrum-based fault localization utilizes dynamic execution information, this technique works in a *static* way: it operates as a batch process where user interaction is minimal. Specifically, after receiving the program spectra, existing fault localization techniques build models based on the provided input, calculate suspiciousness scores and presents user a list of ranked suspicious program elements. In contrast, human debugging often operates in an *interactive* way: when a developer observes an abnormal execution, she checks code that might be responsible, tries to fix the problem, and re-executes the program. If the problem still exists, she can *rectify* her previous judgment on suspicious code and continue debugging, until the root cause is found.

Inspired by manual debugging processes and the drawbacks of existing spectrum-based fault localization techniques, we pose the following key research question:

> *Can we perform spectrum-based fault localization in an interactive way to improve accuracy while incurring little additional cost?*

In this paper, we propose a generic framework TALK, which incorporates user feedback to spectrum-based fault localization approaches. Each time a user inspects a suspicious program element recommended by a fault localization method, the user can judge the correctness of the recommended element and provide this information as feedback to TALK. Based on the feedback, our proposed

The work was done while the first author was visiting Singapore Management University.

978-1-4673-2312-3/12/$31.00 © 2012 IEEE

Statement		*Test case*												*Suspiciousness Metrics*						
		t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12	N_{ef}	N_{ep}	N_{nf}	N_{np}	*Ochiai*	*Tarantula*	*Jaccard*
`main(){`	s1	●	●	●	●	●	●	●	●	●	●	●	●	4	8	0	0	0.577	0.500	0.333
`int let, dig, other, c;`	s2																			
`let = dig = other = 0;`	s3																			
`while(c=getchar()){`	s4																			
`if('A'<=c && 'Z'>=c)`	s5	●	●	●	●	●	●	●		●	●	●	●	4	7	0	1	0.603	0.533	0.364
`let += 1;`	s6	●		●	●	●	●	●			●	●	●	3	6	1	2	0.500	0.500	0.300
`else if('a'<=c && 'z'>=c)`	s7	●	●	●	●	●	●	●		●	●	●	●	4	7	0	1	0.603	0.533	0.364
`let += 1;`	s8		●		●	●		●				●		3	2	1	6	0.671	0.750	0.500
`else if('0'<=c && '9'>c) /*FAULT*/`	s9	●	●	●		●	●	●		●		●	●	4	5	0	3	0.667	0.615	0.444
`dig += 1;`	s10		●	●			●	●		●		●		2	4	2	4	0.408	0.500	0.250
`else if(isprint(c))`	s11	●	●			●	●	●		●		●	●	4	4	0	4	0.707	0.667	0.500
`other += 1;}`	s12	●	●			●	●	●		●		●		4	3	0	5	0.756	0.727	0.571
`printf("%d %d %d\n",let,dig,others);}`	s13	●	●	●	●	●	●	●	●	●	●	●	●	4	8	0	0	0.577	0.500	0.333
pass/fail		**F**	**F**	**P**	**P**	**F**	**P**	**P**	**P**	**P**	**P**	**F**	**P**							

Figure 1. Running Example

interactive framework will re-order the elements that are yet to be checked, and present an updated list of suspicious elements. Unlike [11], [12], our method simply asks users to judge if a particular program element is correct or not, and thus needs only "simple" feedback from users.

To utilize simple feedback, we leverage the relationships among false positives and potential root causes in program spectra. When a false positive is identified by a user, all program elements that are executed together with the false positive in failed executions should be more suspicious than before as one or a few of them must be the root cause(s) causing the failures. We then adjust supiciousness by 1) infering root cause from false positives and 2) focusing on one failed profile with the least remaining program elements covered. After that, the framework updates the recommended list for further inspection.

We evaluate our framework on five real *C* programs and seven Siemens programs from the Software-artifact Infrastructure Repository (SIR). In total, we analyze 254 faults, and demonstrate that our approach outperforms conventional non-interactive fault localization methods.

The main contributions of this paper are as follows:

1) We propose TALK, a novel framework for interactive SBFL. TALK incorporates user feedback to update the list of suspicious elements, while providing those feedbacks requires no additional effort from user than conventional non-interactive methods.
2) TALK is a one-size-fits-all approach that can be applied to most existing static SBFL techniques.
3) We have evaluated our approach on 12 *C* programs. Our evaluation demonstrates that simple user feedback can help significantly improve the accuracy of existing fault localization approaches.

The rest of this paper is organized as follows: Section II introduces preliminaries on fault localization and describe the motivation of this work. Section III presents our approach in detail. Section IV shows the empirical evaluation and Section V describes related work. Finally, Section VI concludes with future work.

Table I
SPECTRUM-BASED FAULT LOCALIZATION

Name	Suspiciousness Calculation Formula
Tarantula	$\dfrac{\frac{N_{ef}(s)}{N_{ef}(s)+N_{nf}(s)}}{\frac{N_{ef}(s)}{N_{ef}(s)+N_{nf}(s)}+\frac{N_{ep}(s)}{N_{ep}(s)+N_{np}(s)}}$
Ochiai	$\dfrac{N_{ef}(s)}{\sqrt{(N_{ef}(s)+N_{nf}(s))\cdot(N_{ef}(s)+N_{ep}(s))}}$
Jaccard	$\dfrac{N_{ef}(s)}{N_{ef}(s)+N_{nf}(s)+N_{ep}(s)}$

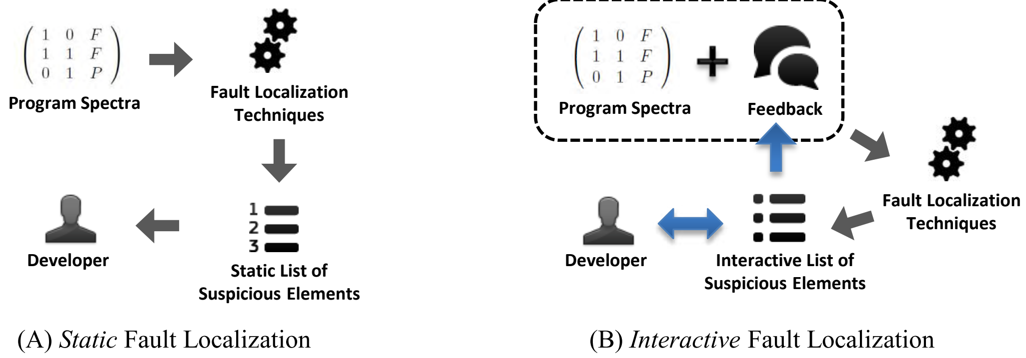
II. Preliminaries and Motivation

A. Fault Localization

Spectrum-based fault localization [1], [16], [19], [20] aims to locate faults by analyzing *program spectra* which often consists of information about whether a program element (*e.g.*, a function, a statement, or a predicate) is hit in an execution. Statistical calculations are then performed on passed and failed executions to obtain the suspiciousness score for every element executed. All elements are then sorted in descending order according to their suspiciousness scores for developers to investigate.

Figure 1 shows a code snippet (taken from [9], [14]) which contains 13 statements $s_1 ... s_{13}$, where s_9 is faulty (the condition `'9'>c` should be `'9'>=c`). Suppose the program has 12 test cases $t_1 ... t_{12}$. A $\bullet$ for a statement under a test case means the corresponding statement is executed (or covered) in the corresponding test case. With the spectra for all of the test cases and their *pass/fail* information, fault localization techniques may calculate various suspiciousness scores for each of the statements and rank them differently.

The key for a spectrum-based fault localization technique is the formula used to calculate suspiciousness. Table I lists the formulae of three well-known techniques: *Tarantula* [16], *Ochiai* [1], and *Jaccard* [1]. Given a program element s, $N_{ef}(s)$ is the number of ***failed*** executions that ***execute*** s, $N_{np}(s)$ is the number of ***passed*** executions that do ***not*** hit s, $N_{nf}(s)$ is the number of ***failed*** executions that do ***not*** hit s,



(A) *Static* Fault Localization (B) *Interactive* Fault Localization

Figure 2. The Framework of *Static* and *Interactive* Fault Localization Techniques.

and $N_{ep}(s)$ is the number of ***passed*** executions that ***execute*** s. In this case, three well-known techniques, *Ochiai* [1], *Tarantula* [16], and *Jaccard* [1] all rank s_{12} as the most suspicious statement (see the last three columns of Figure 1 and compare the suspiciousness score of s_{12} with those of the other statements) and thus produce a *false positive*.

Each column for t_i in Figure 1 is a spectrum. The columns N_{ef}, N_{ep}, N_{nf}, and N_{np} can thus be calculated from the spectra. The suspiciousness scores of *Tarantula*, *Ochiai*, and *Jaccard* for each statement are then calculated based on the formulae in Table I.

B. Problem & Motivation

Conventional non-interactive fault localization techniques work in a *static* way that does not incorporate user feedback. Specifically, existing fault localization techniques (*e.g.*, *Ochiai* and *Tarantula*) often take *program spectra* as input, calculate a suspiciousness score for each *program element* (*i.e.*, function, predicate or statements) and output a ranked list in which program elements are ranked in descending order of their suspiciousness scores. Developers are expected to examine the code along with the ranked list of suspicious program elements. Figure 2 (A) shows the general process.

Although many studies [1], [13], [16] show that spectrum-based fault localization is effective in helping locate faults, the performance of existing spectrum-based fault localization techniques is still unsatisfactory [26]. In some cases, developers have to examine many statements before reaching the real root cause. Those statements are fault-free but for some reasons appear frequently in failed executions and relatively less in passed executions. As a result, these statements are marked as highly suspicious by conventional non-interactive spectrum-based fault localization (SBFL) techniques but are identified as *false positive*s after they are examined by developers. User feedback in the form of the identification of false positives could be collected one by one as users investigate the sorted list of suspicious program elements. These *false positive*s indicate the imprecision of existing fault localization approaches and thus collecting them might be helpful for improving these techniques. This prompts us to investigate the following question which is an instantiation of the question introduced in Section I:

> *If we can ask a user to mark examined program elements as **faulty** or **clean** and record these feedback one at a time, how can we update a fault localization approach to improve the accuracy of the fault localization process for the remaining uninvestigated program elements?*

C. Solution Intuition

To answer the question posed in the previous section, in this section we introduce a simple scenario that illustrates our *interactive* fault localization approach.

The difficulty of incorporating user feedback resides in the SBFL models. Existing fault localization techniques (such as *Tarantula*, *Jaccard* and *Ochiai*) calculate the suspiciousness of each program element separately (see Figure 1, Table I) and consequently establish no connections between program elements. Thus, user marked labels on examined elements cannot be used directly to adjust the suspiciousness of remaining uninspected elements. Associations among statements are needed to incorporate user feedback so that the label of one examined statement (faulty or clean) can be linked to the suspiciousness of other uninspected ones.

Investigating *co-occurrences* of program elements in failed executions is one way to build such association and help adjust existing fault localization techniques. Let $\mathcal{D}$ represents the set of all elements covered by program spectra and an execution profile be represented as $t_i = \langle e_{i1}, e_{i2}, ..., e_{i|\mathcal{D}|} \rangle$, $e_{ij} = 1$ if the jth program element is covered by the profile t_i; otherwise $e_{ij} = 0$. Suppose, in a failed profile t_i, only two elements s_{x_1} and s_{x_2} are covered (*i.e.*, $e_{ij} = 1$ *iff* $j = x_1 \land j = x_2$). If a user checks s_{x_1} and finds that s_{x_1} does not contain a fault, under the assumption that the collected profile is accurate, we are highly confident that s_{x_2} is faulty[1], no matter what information is provided by other execution profiles. In this case, we can transfer the suspiciousness score of s_{x_1} to that of s_{x_2} so that s_{x_2} will be considered as the most promising

[1]In some cases this is not necessarily true, *e.g.*, omission errors [31].

candidate for inspection. Generalizing this scenario, if a failed profile t_i covers multiple unchecked statements, we split the suspiciousness score of a checked clean statement to those unchecked statements.

Consider Figure 1 as an example. After inspecting s_{12} and s_{11} following the list recommended by *Ochiai*, a developer will identify them as *clean*. According to the initial rank list, the user will first check s_8 followed by s_9, as the former gets a relatively higher suspiciousness score. However, among all failed executions that cover s_{12} and s_{11} (*i.e.*, t_1, t_2, t_5 and t_{11}), s_9 appears 4 times (thus, all of them) while s_8 appears 3 times (not covered by t_1). In this case, using our interactive approach, s_8 gets less shares from the suspiciousness scores of s_{11} and s_{12}, thus s_8 is less likely than s_9 to be the root cause of these four failures—indeed s_8 is not the root cause of t_1's failure, as it is not covered in that execution profile.

III. Fault Localization based on User Feedback

In this section we introduce our approach: *Spectrum-based Faul**T** Loc**A**lization **L**everaging User Feedbac**K*** (abbr. TALK). TALK leverages user feedback to adjust suspiciousness scores of program elements during fault localization process. In the subsections, we show interactive SBFL framework and describe the details of our approach.

A. Framework Utilizing User Feedback

Figure 2 shows the differences in debugging process between conventional *static* fault localization methods and our *interactive* approach considering user feedback. The overall process of TALK is shown in Figure 2(B) in which a spectrum-based fault localization model f (*e.g.*, *Ochiai*) recommends suspicious program elements in an *iterative* way. In the first round, f takes in a program spectra T, builds a model f_T based on the input T, calculates a suspiciousness score $f_T(s)$ for each program element s using the model and outputs a ranked list in descending order of their suspiciousness scores. Developers are then expected to inspect one or a set of program elements S_i (i is the iteration number) at the top of the ranked list of program elements and label each element in S_i (*i.e.*, either *clean* or *faulty*). Let those statements and corresponding labels be denoted as $\mathcal{F}_i$. Having received the label set $\mathcal{F}_i$, our framework will incorporate $\mathcal{F}_i$ to update the current fault localization model $f_T \rightarrow f_{(T, \bigcup_i \mathcal{F}_i)}$ and recalculate the suspiciousness of each uninspected program elements by the new model. It then ranks the program elements according to their new suspiciousness scores and outputs a new ranked list for developer's inspection. In the next iteration, the developer can continue to provide more feedback if they are not satisfied with the current recommendation.

Illustrated in Figure 3, our proposed interactive approach provides options for the developer to mark the label of program elements as *clean* or *faulty*. After a developer commits the label(s), our framework will incorporate the labels and update the existing fault localization model. A newly ranked list is then generated. The interactive user interface will update and show the new result which only contains uninspected program elements.

No.	Susp.	Program Element	Buggy?
s_{12}	0.756	`other += 1;}`	✔ ✖
s_{11}	0.707	`else if(isprint(c))`	✔ ✖
s_8	0.671	`let += 1;`	✔ ✖
s_9	0.667	`else if('0'<=c && '9'>c)`	✔ ✖
s_5	0.603	`if('A'<=c && 'Z'>=c)`	✔ ✖

Commit

Figure 3. Interactive Ranked List of Suspicious Program Elements

Our framework in general allows developers to label any element in the ranked list, not just the top ones. Of course, the effectiveness of the feedbacks from labelling different elements would be different. Our study assumes developers always label the top elements, and leave the investigation of different labeling strategies of users as future work.

B. Suspiciousness Updating Rules

Rule R1: Identifying a Root Cause from Its Symptom. Some program elements are deemed highly suspicious based on their coverage profile, as they appear frequently in failed executions and relatively less in passed executions. However, in many cases, those program elements are not faulty and thus are identified as *false positive*s after inspection. A *false positive* is triggered by its root cause in a certain scenario. For example, in Figure 1, s_{12} is covered in all failed executions, but appears in a few passed executions. Consequently it is considered as the most suspicious element by *Ochiai*, *Jaccard* and *Tarantula*. However, s_{12} does not contain a fault. Analyzing the code snippet, we can find that s_9 has been seeded a fault(`'9'>=c` $\rightarrow$ `'9'>c`) so that for each input with a character `'9'`, s_9 will mistakenly direct the control flow to s_{11}. Since character `'9'` is printable, the predicate in s_{11} is always *true*, which leads to the execution of s_{12}. As a result, s_{12} is covered in every failed execution profiles and gets a high suspiciousness score. Statements like s_{12} are only a manifestation of an incorrect program behavior, rather than the root cause of failures, and thus we call it a *symptom*. Symptoms are triggered by their root causes and are mere false positives.

In our method, we adjust the suspiciousness of a possible root cause of a *symptom*. To identify the root cause (*e.g.*, s_9) of the symptom (*e.g.*, s_{12}), we need to associate related program elements together. Under this model, when *false positive*s are identified, we can adjust the suspiciousness scores of other related program elements. Here is our approach to update suspicious scores when a program element s is considered as *clean*:

First of all, we identify the possible root cause of the *symptom* s. We isolate the set of all failed profiles that cover

s—denoted as $T_{fail}(s)$ and calculate the likelihood of an element s_c to be the root cause by the following equations:

$$\mathcal{P}_s(s_c) = \sum_{t \in T_{fail}(s) \wedge s_c \in t} \frac{|\mathcal{D}|}{|\{s' | s' \in t \wedge s' \notin \mathcal{I}\}|} \quad (1)$$

where $s_c \in t$ means that the element s_c is covered by the profile t. Set $\mathcal{D}$ includes all elements executed in the input program spectra T and set $\mathcal{I}$ contains all the elements that have already been inspected. With this equation, we give a score to each element indicating the number of co-occurrences of s and s_c in failed profiles weighted by the lengths of the failed profiles, where the length of a profile is the number of elements covered by the execution profile. Profiles with fewer elements will contribute more to this likelihood score.

Then, the possible root cause s_r of a symptom s is selected in the following equation. Intuitively speaking, we select an uninspected statement as the root cause if it coappears most frequently with the symptom in failed traces covering limited number of uninspected program elements.

$$s_r \leftarrow \arg\max_{s_c \in T} \{\mathcal{P}_s(s_c)\} \quad (2)$$

Next, we want to adjust the suspiciousness score of s_r. Let us denote all profiles that cover program element s_r as $T(s_r)$ and profiles that do not cover program element s_r as $T(\overline{s_r})$. Let us build the fault localization models $f_{T(s_r)}$ and $f_{T(\overline{s_r})}$ based on each of the two profile sets respectively. $f_{T(s_r)}(s_i)$ means the suspiciousness score of statement s_i in model $f_{T(s_r)}$; likewise $f_{T(\overline{s_r})}(s_i)$ is the suspiciousness score of s_i in model $f_{T(\overline{s_r})}$. Then we adjust the suspiciousness score of s_r as follows:

$$Susp_{s_r} = f_T(s_r) + \mathcal{W}_{s_r \to s} \cdot f_T(s) \quad (3)$$

$$\mathcal{W}_{s_r \to s} = f_{T(s_r)}(s) - f_{T(\overline{s_r})}(s) \quad (4)$$

In the equation, s is the program element examined by the user and is identified as *clean*(*i.e.*, the symptom). To update the suspiciousness score of s_r, we contribute a part of the suspiciousness score of s to s_r based on the difference between $f_{T(s_r)}(s)$ and $f_{T(\overline{s_r})}(s)$. The intuition is straightforward: if s_r is the root cause for these failed executions, then the suspiciousness of s in program spectra which covers s_r should be higher than the suspiciousness of s in spectra where s_r is not executed. Otherwise if s_r is not the root cause, $\mathcal{W}_{s_r \to s}$ should intuitively be small.

Note that this rule only adjusts the suspiciousness score for one program element that is most likely to be the root cause for each symptom. We do this because there are often a substantial number of elements that are coincidentally executed in failed executions which might seem to be the root causes, but real faults in the program may often involve only a limited number of program elements. We thus only adjust the suspiciousness for one element to minimize the impact of coincidental candidates.

Rule R2: Focusing on a Single Failed Execution Profile. This heuristic makes the fault localization model focus on only one failed execution. Suppose there are two *failed* execution profiles t_1 and t_2: t_1 covers 5 program elements, and t_2 covers 50 program elements. As the root cause must be executed in a failed profile, focusing on the profile with the least number of elements covered by it will help locate faults more easily. In this example, focusing on elements executed by t_1 is better because at most 5 statements have to be inspected to locate the root cause.

In our approach, we scan all profiles and find out the failed profile t_{min} covering the least number of unexamined elements. For each program element s_i that is covered in t_{min}, we adjust their suspiciousness scores as follows:

$$Susp'_{s_i} = \mathcal{K}_{s_i} \cdot Susp_{s_i} \quad (5)$$

where $\mathcal{K}_s$ is a constant that guarantees $\forall s_i \in t_{min}, s_j \notin t_{min}\ Susp'_{s_i} > Susp'_{s_j}$. In the experiment, we set $\mathcal{K}_s = 10$ (any other values that are larger than 10 is fine) when element s is covered by t_{min}, otherwise $\mathcal{K}_s = 1$.

In this rule, feedback is taken into consideration when multiple faulty lines exist. To focus on remaining undiscovered faulty lines, we also exclude the profiles covering the faulty lines that have already been identified by prior user feedback when choosing t_{min} (see the pseudocode in Figure 4 and its explanations in Section III-C)

C. Overall Approach

The pseudocode of ***TALK*** is shown in Figure 4. To start the process, at lines 1-3 we first build a fault localization model f_T based on the input program spectra T only. Procedure ***show_result***($\mathcal{L}$) presents the ranked list of suspicious program elements $\mathcal{L}$ to the user. $\mathcal{L}_\mathcal{T}$ is an internal data structure that stores a temporary ranked list of program elements. Procedure ***wait_feedback()*** returns *true* if a user submits a new feedback. ***obtain_feedback()*** returns the new feedback as a set of $\langle element, label \rangle$ pairs.

If a statement does not contain a fault (*i.e.*, the statement is a *symptom*), at lines 10-12, we identify its root cause and update the suspiciousness score of the root cause according to Equation 3. The method will record a faulty element if it is identified as such by the user (line 13-14).

At lines 16-18, we adjust the suspiciousness scores of the program elements following the the second rule. To focus on remaining defects in both single and multiple faulty lines cases, procedure ***least_fail_profile($\mathcal{E}$)*** returns the failed profile that 1) covers the least number of elements uninspected and 2) does not cover any element in set $\mathcal{E}$.

IV. Empirical Evaluation

In this section we present an empirical evaluation that analyzes the impact of user feedback on existing

Procedure *TALK*
Input:
T - Program spectra
Output:
$\mathcal{L}$ Ranked list of suspicious program elements
Method:
1: Build a fault localization model f_T using T
2: Obtain a ranked list $\mathcal{L}_\mathcal{T}$ according to f_T
3: $\mathcal{L} \leftarrow \mathcal{L}_\mathcal{T}$; ***show_result***($\mathcal{L}$)
4: **while** ***wait_feedback()*** **do**
5: $\mathcal{F} \leftarrow$ ***obtain_feedback()***
6: **for each** $\langle s, l_s \rangle \in \mathcal{F}$ **do**
7: $\mathcal{L}_\mathcal{T} \leftarrow \mathcal{L}_\mathcal{T} \setminus s$; $\mathcal{I} \leftarrow \mathcal{I} \bigcup \{s\}$
8: **if** l_s =*clean* **then**
9: *//Identify the root cause*
10: $\forall s_c \in T$, calculate $\mathcal{P}_s(s_c)$ by Equation 1
11: Select s_r by Equation 2
12: Update $Susp_{s_r}$ in $\mathcal{L}_\mathcal{T}$ according to Equation 3
13: **else if** l_s =*faulty* **then**
14: $\mathcal{E} \leftarrow \mathcal{E} \bigcup \{s\}$
15: **end if**
16: $\mathcal{L} \leftarrow \mathcal{L}_\mathcal{T}$
17: *//Focus on one failed profile*
18: $t_{min} \leftarrow$***least_fail_profile($\mathcal{E}$)***
19: Update $Susp_{s_r}$ in $\mathcal{L}$ according to Equation 5
20: ***show_result***($\mathcal{L}$)
21: **end for**
22: **end while**
23: return $\mathcal{L}$

Figure 4. TALK: The Proposed Approach

spectrum-based fault localization methods, and compares the diagnostic costs of conventional non-interactive SBFL and TALK. Section IV-A describes the experimental setup in detail and gives descriptive statistics of the subject programs. Section IV-B presents the research questions investigated in our study. Section IV-C shows the results. Finally, Section IV-D discusses the threats to validity.

A. Experimental Setup

Evaluation Metric. We compare the effectiveness of different fault localization methods based on diagnostic cost that calculates the percentage of statements examined to locate a fault, which is commonly used in the literature [1], [16], [22]. The diagnostic cost is defined as follows:

$$Cost = \frac{|\{j \mid f_{T_S}(d_j) \geq f_{T_S}(d_*)\}|}{|\mathcal{D}|}, \qquad (6)$$

where $\mathcal{D}$ consists of all program elements appearing in the input program. We calculate the cost as the percentage of elements that developers have to examine until the root cause of the failures (d_*) are found. Since multiple program elements can be assigned with the same suspicious score, the numerator is considered as the number of program elements d_j that have larger or equal suspicious scores as that of d_*.

Evaluation Assumption. To measure the effectiveness of interactive fault localization methods with user feedback, we assume that:

1) User inspects program elements following recommended list from the most suspicious to the least.
2) Although our framework provides an option for a user to submit either no or multiple feedback. For the simplicity of evaluation, we assume that a feedback is submitted after each program element is inspected and a user will keep labeling until faults are found.

Experimental Dataset. We use five real *C* programs and seven Siemens test programs from the *Software-artifact Infrastructure Repository* (SIR) [8]. We refer to the five real programs (`sed`, `flex`, `grep`, `gzip`, and `space`) as UNIX programs. Table II shows the descriptive statistics of each subject program, including the number of faults, available test cases and code sizes. In these subjects, different types of bugs are seeded (*e.g.*, wrong conditional expressions, additional statements, wrong return expressions, etc.). Following existing studies [1], [15], we exclude faults not directly observable by the profiling tool we use[2]. Subjects are instrumented in statement level and in total we study 254 faults.

Table II
SUBJECT PROGRAMS

Program	Description	LOC	Tests	Faults
tcas	Aircraft Control	173	1609	41
schedule2	Priority Scheduler	374	2710	8
schedule	Priority Scheduler	412	2651	8
replace	Pattern Matcher	564	5543	31
tot_info	Info Measure	565	1052	22
print_tokens2	Lexical Analyzer	570	4055	10
print_tokens	Lexical Analyzer	726	4070	7
space	ADL Compiler	9564	1343	30
flex	Lexical Parser	10124	567	43
sed	Text Processor	9289	371	22
grep	Text Processor	9089	809	17
gzip	Data Compressor	5159	217	15

B. Research Questions

RQ1: *Is user feedback helpful for improving fault localization accuracy?*

TALK is built upon conventional non-interactive fault localization techniques by incorporating user feedback. We would like to compare the accuracies of fault localization before and after user feedback are incorporated. We created *Ochiai*$^+$, *Jaccard*$^+$ and *Tarantula*$^+$ which are the interactive versions of *Ochiai*, *Jaccard* and *Tarantula*, using our proposed TALK framework, respectively.

RQ2: *What is the relative effectiveness of the two rules described in Section III for improving fault localization?*

In Section III-B, we propose two rules to update suspiciousness scores of program elements: 1) identifying

[2] http://gcc.gnu.org/onlinedocs/gcc/Gcov.html

a root cause from its symptom; 2) focusing on a single failed profile with the least number of uninspected elements. To answer RQ2, we simply compare the diagnostic costs of three variants of *Ochiai*$^+$, *Jaccard*$^+$, and *Tarantula*$^+$ respectively: one that uses only the first rule, another that uses only the second rule, and the third that uses both rules.

C. Experimental Results

This section presents our experimental results by addressing the research questions.

RQ1: *Is user feedback helpful for improving fault localization accuracy?*

Following [4], [5] and the *Cost* metric (Equation 6), we compare the effectiveness of two fault localization techniques $\boldsymbol{f}^+$ and $\boldsymbol{f}$ by using one of the methods ($\boldsymbol{f}$) as the reference measure. The Cost difference: $Cost(\boldsymbol{f}) - Cost(\boldsymbol{f}^+)$ is considered as the improvement of $\boldsymbol{f}^+$ over $\boldsymbol{f}$. A positive value means that $\boldsymbol{f}^+$ performs better than $\boldsymbol{f}$ (since a lower Cost is preferred). The difference corresponds to the magnitude of improvement. For example, if the diagnostic cost utilizing $\boldsymbol{f}^+$ is 30% and the diagnostic cost utilizing $\boldsymbol{f}$ is 40%, then the improvement of $\boldsymbol{f}^+$ over $\boldsymbol{f}$ is 10%, which means that developers would examine 10% fewer statements if $\boldsymbol{f}^+$ is deployed.

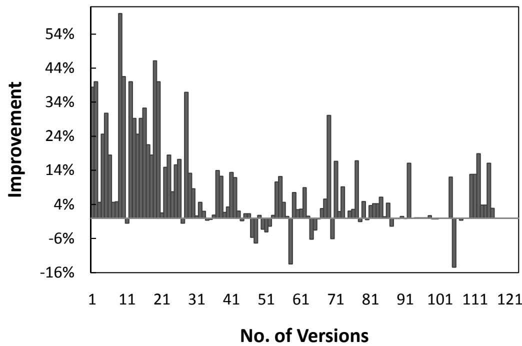


Figure 6. Improvement of *Ochiai*$^+$ over *Ochiai* on all dataset.

Figures 6, 7, and 8 show the comparisons of diagnostic costs between different fault localization techniques. The horizontal axes represent the versions that show differences in the diagnostic costs. The vertical axes represent the percentage difference in the diagnostic costs. If our proposed methods are better than the corresponding conventional non-interactive fault localization methods, the area above the horizontal axes will be larger. In these figures we show only versions for which there are diagnostic cost differences.

***Ochiai*$^+$ vs *Ochiai*.** Figure 6 compares the diagnostic cost utilizing *Ochiai*$^+$ and the diagnostic cost utilizing *Ochiai* over all faulty versions of Siemens and UNIX programs. *Ochiai* is used as the reference fault localization technique. Each program version is a bar in this graph and we remove versions from the graph that have no diagnostic cost differences due to the limited space. The vertical axis

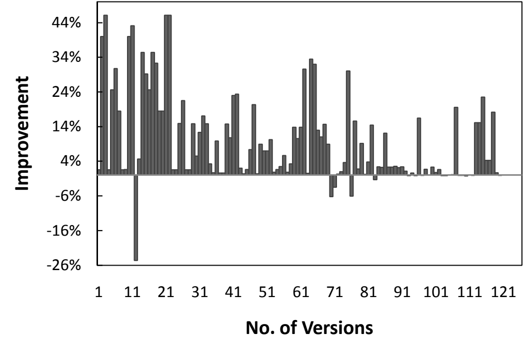


Figure 7. Improvement of *Jaccard*$^+$ over *Jaccard* on all dataset.

represents the magnitude of improvement of *Ochiai*$^+$ over *Ochiai*. If the bar of a faulty version is above the horizontal axis, that means on this version *Ochiai*$^+$ performs better than *Ochiai* (positive improvement). However, bars below the horizontal axis represent faulty versions for which *Ochiai*$^+$ performs worse than *Ochiai*.

The result shows that in general *Ochiai*$^+$ is better than *Ochiai*. Out of 117 versions that show differences in diagnostic cost, *Ochiai*$^+$ performs better than *Ochiai* on 83 versions, and only performs worse on 34 versions. The positive improvement ranges from 0.06% to 60.0%, with an average of 12.29%.

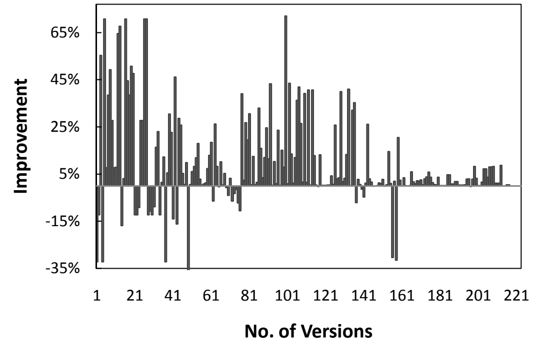


Figure 8. Improvement of *Tarantula*$^+$ over *Tarantula* on all dataset.

***Jaccard*$^+$ vs *Jaccard*.** Figure 7 compares the diagnostic cost utilizing *Jaccard*$^+$ and the diagnostic cost utilizing *Jaccard* over all faulty versions of Siemens and UNIX programs. *Jaccard* is used as the reference fault localization technique. The result shows that in general *Jaccard*$^+$ is better than *Jaccard* on Siemens and UNIX programs. Out of the 120 versions that show differences in diagnostic cost, *Jaccard*$^+$ performs better than *Jaccard* on 105 versions, and only performs worse on 15 versions. The positive improvement ranges from 0.02% to 46.15%, with an average of 11.71%.

***Tarantula*$^+$ vs *Tarantula*.** Figure 8 compares the diagnostic cost utilizing *Tarantula*$^+$ and the diagnostic cost utilizing *Tarantula* over all faulty versions of Siemens and UNIX programs. *Tarantula* is used as the reference fault localization technique. The result shows that in general *Tarantula*$^+$

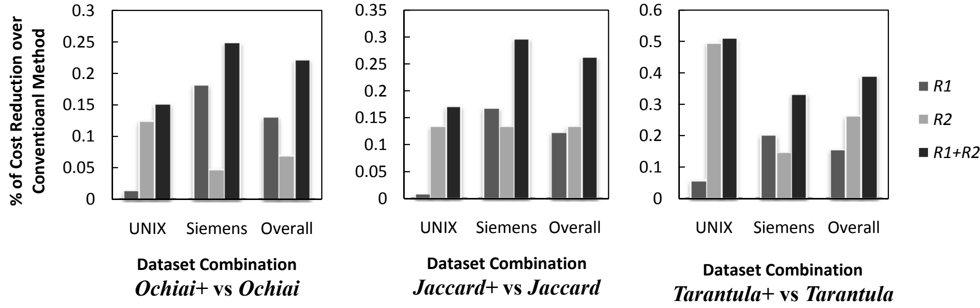


Figure 5. The contributions of different rules.

is better than *Tarantula* on Siemens and UNIX programs. Out of the 217 versions that show differences in diagnostic cost, *Tarantula*$^+$ performs better than *Tarantula* on 178 versions, and only performs worse on 39 versions. The positive improvement ranges from 0.04% to 71.99%, with an average of 13.37%.

Summary. Tables III and IV summarize the results of comparing interactive fault localization approaches with conventional fault localization approaches. We notice that for each conventional approach (*Ochiai*, *Jaccard*, and *Tarantula*) incorporating user feedback with TALK improves the accuracy of the corresponding reference approach.

Table III
COMPARISON OF FAULT LOCALIZATION METHODS.

Fault Loc. Tech.	Positive	Negative	Neutral
Ochiai$^+$ vs *Ochiai*	**31.68%**	12.98%	55.34%
Jaccard$^+$ vs *Jaccard*	**40.08%**	5.73%	54.20%
Tarantula$^+$ vs *Tarantula*	**67.94%**	14.89%	17.18%

Ochiai has been shown to be the most accurate among the three conventional approaches. From the first row of Table III, we note that *Ochiai*$^+$ performs better than *Ochiai* on 31.68% of the faulty versions, worse on 12.98% of the faulty versions, and shows no improvement on 55.34% of the faulty versions. The first row of Table IV characterizes the degree of positive improvement of *Ochiai*$^+$ over *Ochiai*. Half of the 31.68% faulty versions with positive improvement values have improvements between 0.06% and 7.50%, and the other half have improvements between 7.50% and 60.0%. The average positive improvement is 12.29%.

Table IV
DISTRIBUTION OF POSITIVE IMPROVEMENTS.

Fault Loc. Tech.	Max	Mean	Median	Min
Ochiai$^+$ vs *Ochiai*	**60.0%**	12.29%	7.50%	0.06%
Jaccard$^+$ vs *Jaccard*	**46.15%**	11.71%	7.00%	0.02%
Tarantula$^+$ vs *Tarantula*	**71.99%**	13.37%	4.07%	0.04%

To further confirm the effectiveness of TALK, we also conduct paired *Wilcoxon signed-rank test* [28] to check if the differences in the mean diagnostic costs of *Ochiai*$^+$ and *Ochiai*, *Jaccard*$^+$ and *Jaccard*, and *Tarantula*$^+$ and *Tarantula* over the entire dataset are significant or not. The statistical test results reject the null hypotheses and suggests that *Ochiai*$^+$, *Jaccard*$^+$, and *Tarantula*$^+$ are significantly better than *Ochiai*, *Jaccard*, and *Tarantula* respectively, at 95% confidence interval.

Our interactive approach (*Ochiai*$^+$) improves *Ochiai* by reducing the amount of inspected code by up to 3047 lines of code (LOC) for a bug. Among 40% of buggy versions where there is a difference between our approach and *Ochiai*, we reduce the amount of inspected code by at least 50 LOC. Half of the bugs require examining less than 69 LOC using our approach, while *Ochiai* requires less than 81 LOC. *Ochiai*$^+$ successfully localizes 78 bugs by investigating at most 30 lines of code. *Ochiai* only successfully localizes 68 bugs.

We note that user feedback sometimes hurts rather than helps. TALK considers only simple feedback that labels the correctness of a statement, and uses a heuristic to decide the root cause of a bug which may be inaccurate with limited information. For example, some correct statements may happen to co-appear frequently with symptoms. For a minority of cases, the inaccuracy of this heuristic causes user feedback to hurt rather than help. But as we can see in Figure 6, 7, and 8, most of the negative improvements of those versions are relatively small when compared to the positive improvements.

RQ2: *What is the relative effectiveness of the two rules described in Section III for improving fault localization?*

Figure 5 shows the contributions of each rule to the performance of *interactive* fault localization. Each bar in the figure represents the percentage of diagnostic cost reduction when either one or both of the rules are employed. For example, applying Rule *R1* alone to *Ochiai* on Siemens programs incurs an average cost of 20.98% while *static Ochiai* incurs an average cost of 25.63%. So the cost reduction is $\frac{25.63\%-20.98\%}{25.63\%} = 18.14\%$.

As shown in the figure, different rules contribute differently to the effectiveness of the overall approach (which incorporates both rules) on different datasets. For example, Rule *R1* alone reduces the average cost by 1.4% and 18.14% on UNIX and Siemens programs respectively. Rule *R2* alone reduces the average cost by 12.34% and 4.74% on UNIX and Siemens programs respectively. Thus, *R1* contributes more to the accuracy of the overall approach

on Siemens programs, and *R2* contributes more on UNIX programs. This is probably due to the differences in program sizes and types of seeded faults. Specifically, UNIX programs are real-life programs which contain thousands of lines of code. Focusing on a single failed profile (Rule *R2*) will help to avoid examining a substantial amount of code. Conventional methods perform poorly on Siemens programs (*e.g.*, 25.63% average cost using *Ochiai*) and thus a lot of false positives can be found and reduced by feedback. This manifests the usefulness of Rule *R1* (inferring a root cause from its symptom). Overall, when all rules are applied, the performance is the best (an average reduction of 15.1% on Siemens programs and 24.8% reduction on UNIX programs). Conclusively, both rules contribute to the effectiveness of *interactive* fault localization.

D. Threats to Validity

Threats to construct validity refers to the suitability of our evaluation metric. We use a cost metric that has been utilized to evaluate past fault localization techniques [4], [5]. We believe this is a fair and well-accepted metric.

Threats to external validity refers to the generalizability of our findings. In this study, we have evaluated our approach on not only Siemens programs but also five real programs. All of our programs, however, are written in *C*. In the future, we plan to investigate more programs written in various programming languages and to explore more strategies for utilizing user feedback to enhance our approach.

Another threat to validity relates to the effect if users are not always correct in deciding the label of inspected program elements. User feedback is just one of the two labels: *buggy* or *clean*. A user mistake would either be 1) considering a correct statement as *buggy*, or 2) considering a faulty statement as *clean*. The user would need to fix the "bug" after making a mistake of the first type, which would require further careful code examination, and the user would eventually realize the mistake. Also, to the best of our knowledge, all fault localization techniques are evaluated by assuming a user is always correct when ascertaining if a program element is buggy or correct.

V. Related Work

In this section, we describe related work on fault localization and defect prediction. The survey here is by no means a complete list.

Fault Localization. Over the past decade, many automatic fault localization and debugging methods have been proposed. Renieris and Reiss propose a nearest neighbor fault localization tool called WHITHER [27] that compares failed executions to correct executions and reports the most suspicious locations in the program. Zeller applies *Delta Debugging* to search for the minimum state differences between a failed execution and a successful execution that may cause the failure [29]. Liblit *et al.* profile predicates, measure the correlations between the true evaluation of the predicates and program failures [18], and consider the predicates with high correlations to be the root causes. *Tarantula* [15], [16] and *Ochiai* [3] analyze spectra and compute suspiciousness of various program elements with different formulae (see Table I).

Although fault localization techniques do not handle all kinds of bugs, they are shown to be useful for debugging (*e.g.*, [26]). Even though bugs may not be exactly located, such tools help to point developers to the right direction. This paper aims to improve the accuracy of existing techniques by incorporating simple user feedback; it doesnt propose an independent fault localization technique or address the general problems faced by fault localization techniques (*e.g.*, multi-threaded, nondeterministic, multiple bugs), which we leave for future work.

The closest to our work, is the work by Hao *et al.* which propose a method to interactively recommend checking points (or break points) [11]. Their method requires both coverage profiles and more detailed traces containing the order of various program elements that are executed. The latter would be very expensive to collect especially on large real programs. In their approach, developers are asked to build connections between false positives and potential faults by determining whether the fault is executed before or after the checking point. Although debugging tools exist (*e.g.*, [23]), providing this kind of feedback is often a difficult and error-prone task. Different from their approach, our method only requires coverage profile and simply asks users to judge if a particular program element is clean or buggy which is a simple feedback. In standard (*i.e.*, static) fault localization approaches, users would also need to examine recommended elements to remove the false positives and find the faults.

Lucia *et al.* adopt user feedback for clone-based bug detection approaches [21]. Their method extracts code features by static analysis and incorporates user feedback to automatically refines a classification model for identifying anomalies and re-sorts the remaining of the reports.

In [12], Insa *et al.* propose a strategy for algorithmic debugging which interatively selects nodes from a execution tree and asks user questions concerning program state. They also prove that their method is optimal for algorithmic debugging. However, answering questions about correct program variable value when a program is at a particular state is not an easy task for medium-sized or large programs.

Defect Prediction. Defect prediction is similar to fault localization but often considers a coarser level of granularity. Many such studies [24], [25] use code sizes, complexity metrics, and variations of various variables, *e.g.*, developer's experience, etc, in software development processes to build prediction models. For example, Nagappan *et al.* [24] use regression models with principal component analysis on the code metrics to predict post-release defects. In [25], Ostrand *et al.* utilize information about the relationships between

programmer and artifacts to predict defects. In [30], Zhang *et al.* propose methods that construct defect prediction models based on a small number of randomly sampled program files. In [17], a new method that actively selects sample files for defect prediction is proposed.

VI. Conclusion and Future Work

This paper proposes a novel method TALK for fault localization, which aims to improve fault localization accuracy by leveraging user feedback while limiting the additional manual cost incurred. In comparison with existing non-interactive spectrum-based fault localization techniques on 12 *C* programs, we have shown that our proposed method can reduce diagnostic cost by up to 71.99% in up to 67.94% of the buggy program versions. Using Wilcoxon rank sum test, we have also shown that our approach statistically significantly outperforms conventional approaches.

Future work includes trying different strategies (*e.g.*, a different formula for identifying root causes) to further utilize user feedback, using different experimental setups (*e.g.*, when a user provides multiple feedbacks in various labeling orders rather than one by one following the recommended list), and evaluating in scenarios where multiple bugs exist. It would also be interesting to enhance TALK by allowing users to rollback their feedback if they made mistakes.

Acknowledgement

This work is partially supported by NSFC grant 61073006 and Tsinghua University project 2010THZ0. We would like to thank the researchers who make the Siemens Test Suite available, and researchers maintaining Software-artifact Infrastructure Repository (SIR) for making the other *C* programs available. We would also like to thank the anonymous reviewers for providing us constructive comments and suggestions.

References

[1] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. van Gemund, "A practical evaluation of spectrum-based fault localization," *Journal of Systems and Software*, 2009.
[2] R. Abreu, "Spectrum-based fault localization in embedded software." Ph.D. dissertation, Delft University of Technology, 2009.
[3] R. Abreu, P. Zoeteweij, and A. J. van Gemund, "Spectrum-Based Multiple Fault Localization," in *ASE*, Auckland, New Zealand, 2009.
[4] G. K. Baah, A. Podgurski, and M. J. Harrold, "Causal inference for statistical fault localization," in *ISSTA*, 2010, pp. 73–84.
[5] ——, "Mitigating the confounding effects of program dependences for effective fault localization," in *SIGSOFT FSE*, 2011, pp. 146–156.
[6] B. Beizer, *Software Testing Techniques*, 2nd ed. Boston: International Thomson Computer Press, 1990.
[7] J. S. Collofello and S. N. Woodfield, "Evaluating the effectiveness of reliability-assurance techniques," *Journal of Systems and Software*, vol. 9, no. 3, pp. 191–195, 1989.
[8] 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.
[9] A. González-Sanchez, É. Piel, R. Abreu, H.-G. Groß, and A. J. C. van Gemund, "Prioritizing tests for software fault diagnosis," *Softw., Pract. Exper.*, vol. 41, no. 10, pp. 1105–1129, 2011.
[10] B. Hailpern and P. Santhanam, "Software debugging, testing, and verification," *IBM Systems Journal*, vol. 41, no. 1, pp. 4–12, 2002.
[11] D. Hao, L. Zhang, T. Xie, H. Mei, and J.-S. Sun, "Interactive fault localization using test information," *Journal of Computer Science and Technology*, vol. 24, no. 5, 2009.
[12] D. Insa and J. Silva, "An optimal strategy for algorithmic debugging," in *ASE*, 2011, pp. 203–212.
[13] D. Jeffrey, N. Gupta, and R. Gupta, "Fault localization using value replacement," in *ISSTA*, 2008.
[14] B. Jiang, W. K. Chan, and T. H. Tse, "On practical adequate test suites for integrated test case prioritization and fault localization," in *QSIC*, 2011, pp. 21–30.
[15] J. Jones, M. Harrold, and J. Stasko, "Visualization of test information to assist fault detection," in *ICSE*, 2002.
[16] J. Jones and M. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *ASE*, 2005.
[17] M. Li, H. Zhang, R. Wu, and Z.-H. Zhou, "Sample-based software defect prediction with active and semi-supervised learning," *Autom. Softw. Eng.*, vol. 19, 2012.
[18] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, "Bug isolation via remote program sampling," in *PLDI*, 2003.
[19] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *PLDI*, 2005.
[20] C. Liu, X. Yan, L. Fei, J. Han, and S. P. Midkiff, "SOBER: Statistical model-based bug localization," in *ESEC/FSE*, 2005.
[21] Lucia, D. Lo, and L. Jiang, "Active refinement of clone anomaly reports," in *ICSE*, 2012.
[22] Lucia, D. Lo, L. Jiang, and A. Budi, "Comprehensive evaluation of association measures for fault localization," in *ICSM*, 2010.
[23] N. Matloff and P. J. Salzman, *The Art of Debugging with GDB, DDD and Eclipse*. No Starch Press, Inc, 2008.
[24] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *Proc. of ICSE '06*, New York, NY, USA, 2006, pp. 452–461.
[25] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Programmer-based fault prediction," in *Proc. of PROMISE '10*, 2010.
[26] C. Parnin and A. Orso, "Are Automated Debugging Techniques Actually Helping Programmers?" in *Proc. of ISSTA*, 2011, pp. 199–209.
[27] M. Renieris and S. Reiss, "Fault localization with nearest neighbor queries," in *ASE*, 2003, pp. 141–154.
[28] F. Wilcoxon, "Individual comparisons by ranking methods," in *Biometrics*, 1943, pp. 80–3.
[29] A. Zeller, "Isolating cause-effect chains from computer programs," in *FSE*, 2002, pp. 1–10.
[30] H. Zhang and R. Wu, "Sampling program quality," in *ICSM*, 2010, pp. 1–10.
[31] X. Zhang, S. Tallam, N. Gupta, and R. Gupta, "Towards locating execution omission errors," in *PLDI*, 2007, pp. 415–424.