Will Fault Localization Work For These Failures ? An Automated Approach to Predict Effectiveness of Fault Localization Tools

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

***Abstract*—Debugging is a crucial yet expensive activity to improve the reliability of software systems. To reduce debugging cost, various fault localization tools have been proposed. A spectrum-based fault localization tool often outputs an ordered list of program elements sorted based on their likelihood to be the root cause of a set of failures (i.e., their suspiciousness scores). Despite the many studies on fault localization, unfortunately, however, for many bugs, the root causes are often low in the ordered list. This potentially causes developers to distrust fault localization tools. Recently, Parnin and Orso highlight in their user study that many debuggers do not find fault localization useful if they do not find the root cause early in the list.**

To alleviate the above issue, we build an oracle that could predict whether the output of a fault localization tool can be trusted or not. If the output is not likely to be trusted, developers do not need to spend time going through the list of most suspicious program elements one by one. Rather, other conventional means of debugging could be performed. To construct the oracle, we extract the values of a number of features that are potentially related to the effectiveness of fault localization. Building upon advances in machine learning, we process these feature values to learn a discriminative model that is able to predict the effectiveness of a fault localization tool output. In this preliminary work, we consider an output of a fault localization tool to be effective if the root cause appears in the top 10 most suspicious program elements. We have experimented our proposed oracle on 200 faulty programs from Space, NanoXML, XML-Security, and the 7 programs in Siemens test suite. Our experiments demonstrate that we could predict the effectiveness of fault localization tool with a precision, recall, and F-measure (harmonic mean of precision and recall) of 54.36%, 95.29%, and 69.23%. The numbers indicate that *many* ineffective fault localization instances are identified *correctly*, while only *very few* effective ones are identified *wrongly*.

I. Introduction

Despite the advancement in software tools and processes, bugs are prevalent in many systems. In 2002, it was reported that software bugs cost US economy more than 50 billion dollars annually [34]. Software testing and debugging cost itself is estimated to account for 30-90% of the total labor spent on a project [4]. Thus there is a need to develop automated means to help reduce software debugging cost. One important challenge in debugging is to localize the root cause of program failures. When a program fails, it is often hard to locate the faulty program elements that are responsible for the failure. The root cause could be located far from the location where the failure is exhibited, e.g., the location where a program crashes or produces a wrong output.

In order to address the high cost of debugging in general, and help in localizing root causes of failures in particular, many spectrum-based fault localization tools have been proposed in the literature, e.g., [19], [1], [24]. These tools typically take in a set of normal execution traces and another set of faulty execution traces. Based on these set of program execution traces, these tools assign suspiciousness scores to various program elements. Next, program elements could be sorted based on their suspiciousness scores in descending order. The resultant list of suspicious program elements can then be presented to a human debugger to aid him/her in finding the root cause of a set of failures.

An *effective* fault localization tool would return a root cause at the top of a list of suspicious program elements. Although past studies have shown that fault localization tools could be effective for a number of cases, unfortunately, for many other cases, fault localization tools are not effective enough. Root causes are often listed low in the list of most suspicious program elements. Parnin and Orso pointed out in their user study that many developers do not find fault localization useful if they do not find the root cause early in the list [26]. This *unreliability* of fault localization tools potentially cause many developers to distrust fault localization tools.

In this work, we plan to increase the usability of fault localization tools by building an oracle to predict if a particular output of a fault localization tool is likely to be effective or not. We define an output of a fault localization tool to be effective if the faulty program element or root cause is listed among the top-10 most suspicious program elements. With our tool, the debuggers could be better informed whether he can trust or distrust the output of a fault localization tool run on a set of program execution traces. The following scenarios illustrate the benefits of predicting the effectiveness of a fault localization output:

Scenario 1 - Without Oracle: Tien-Duy had 10 bugs to fix.

1063-6773/13 $26.00 © 2013 IEEE
DOI 10.1109/ICSM.2013.42

He ran a fault localization tool for the 10 bugs. He followed the tool recommendations, however he only found 2 of the 10 recommendations to be effective. He wasted much time following 8 bad recommendations given by the tool.

Scenario 2 - With Oracle: Tien-Duy had 10 bugs to fix. He ran a fault localization tool for the 10 bugs and he had an oracle that can predict which fault localization outputs are likely to be effective. The oracle predicted that 3 outputs are likely to be effective. For 2 out of the 3 outputs, the fault localization outputs are indeed effective and saved Tien-Duy much time. Tien-Duy only wasted time following 1 bad recommendation.

To build the oracle, we extract values of important features from the execution traces and outputs of fault localization tools. These feature values extracted from a training data are then used to build a discriminative model leveraging a machine learning solution. The resultant discriminative model serves as an oracle and could be used to predict the effectiveness of a fault localization tool on other inputs.

We have experimented our approach on 200 faulty versions from NanoXML, XML-Security, Space, and the 7 programs in the Siemens test suite. We investigate a well known spectrum-based fault localization tool namely Tarantula [19] which was also studied by Parnin and Orso [26]. Our experiments show that we can predict whether a fault localization tool is effective or not by a precision, recall, and F-measure (i.e., harmonic mean of precision and recall) of 54.36%, 95.29%, and 69.23%. We also investigate if our tool is effective to help two other fault localization tools, i.e., Ochiai [1], and Information Gain [24], with promising results.

In this work, our contributions are as follows:

1) We define a new research problem namely predicting the effectiveness of a fault localization tool given a set of execution traces. Solving this problem would help developers to better trust the output of a fault localization tool.
2) We present a machine learning framework to tackle the research problem. We propose a novel set of features that are relevant for predicting the effectiveness of a fault localization tool. We build upon and extend a state-of-the-art machine learning solution for the prediction problem by addressing the issue of imbalanced data. The issue of imbalanced data occurs since many outputs of Tarantula are ineffective.
3) We have evaluated our approach on 200 faulty programs from NanoXML, XML-Security, Space, and the 7 programs from the Siemens test suite. We show that we could achieve a precision, recall, and F-measure of 54.36%, 95.29%, and 69.23%. This shows that *many* ineffective and *almost all* effective outputs of Tarantula are detected correctly.

The structure of this paper is as follows. In Section II, we describe preliminary materials on spectrum-based fault localization and an intuition how effectiveness prediction could be solved. In Section III, we present a birds-eye-view of our proposed framework. Section IV outlines what features are extracted from the execution traces and output of the fault localization tool. Section V elaborates our approach to learn a discriminative model using a classification algorithm and how we address the problem of imbalanced data. We present our experiment settings, datasets, and results which answer a number of research questions in Section VI. We discuss related studies in Section VII. We finally conclude and mention future work in Section VIII.

Table I
SPECTRA NOTATIONS

Symbol	Definition
n	Total number of test cases in the test suite
n^e	Number of test cases that executes a program element e
n_s	Number of test cases that pass
n_f	Number of test cases that fail
n_s^e	Number of test cases that execute e and pass
n_f^e	Number of test cases that execute e and fail

II. PRELIMINARIES & PROBLEM DEFN.

In this section, we first introduce fault localization. We then define the problem of effectiveness prediction and give some intuitions on how this could be solved.

A. Fault Localization

Fault localization takes as input a faulty program, along with a set of test cases, and a test oracle. The faulty program is instrumented such that when a test case is run over it, a program spectra is generated. A program spectra records certain characteristics of a particular program run and thus it becomes a behavioral signature of the run [28]. This program spectra could constitute a set of counters which record how many times different program elements (e.g., statement, basic block, etc) are executed in a particular program run [14]. Alternatively, the counter could record a boolean flag that indicates whether a program element is executed or not. The test oracle is used to decide if a particular program run is correct or faulty. Faulty runs or executions are also referred to as failures. Fault localization task is to analyze program spectra of correct and faulty runs with the goal of finding program elements that are the root causes of the failures (i.e., the faults or errors).

Various spectra have been proposed in past studies [14]. In this study, we use *block-hit spectra*; we instrument every block of a program and collect information on which blocks are executed in a run. Block-hit spectra is suitable as all statements in a basic block have the same execution profile. It has also been shown in the literature that the cost of collecting block-hit spectra is relatively low and the resultant spectra could be used for fault localization [1], [14].

Figure 1 shows an example code with several program spectra. The identifiers of the basic blocks are shown in the first column. The statements located in the basic blocks are

Blk ID	Program Elements	T1	T2	T3	T4

1	int count, n;
	Ele *proc;
	List *src_queue, *dest_queue;
	if (prio >= MAXPRIO) /*maxprio=3*/
		●	●	●	●

2	{return;}	●			

3	src_queue = prio_queue[prio];
	dest_queue = prio_queue[prio+1];
	count = src_queue->mem_count;
	if (count > 1) /* Bug*//* supposed : count >=1*/ {
			●	●	●

4	n = (int) (count*ratio + 1);
	proc = find_nth(src_queue, n);
	if (proc) {
			●	●	

5	src_queue = del_ele(src_queue, proc);
	proc->priority = prio;
	dest_queue = append_ele(dest_queue, proc); } }}
			●	●	

	Status of Test Case Execution :	P	P	P	F

Figure 1. Four Block-Hit Program Spectra

shown in the second column. There is a bug in the example code at basic block three; the condition of the if statement should be "count $>=1$" instead of "count >1". Columns 3 to 6 show the program spectra that are produced when four test cases are run. Three of the test cases do not expose the bug, i.e., running them result in correct executions. The fourth test case exposes the bug, i.e., running it result in a faulty execution. A cell marked by a $\bullet$ indicates that a particular basic block is executed when a particular test case is run. An empty cell indicates that a particular basic block is not executed when a particular test case is run.

To identify the faulty program elements (e.g., basic block 3 in Figure 1), we compute the suspiciousness scores of each of the program elements. There are various ways to define suspiciousness. In this work, we primarily consider a well-known suspiciousness score defined by Jones and Harrold, named Tarantula [19]. Considering several notations in Table I, Tarantula's suspiciousness score can be defined as follows:

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

Tarantula considers an element more suspicious if it occurs more frequently in failed executions than in correct executions. Considering the example shown in Figure 1, the suspiciousness score of block 1 is: $\frac{1}{(1+1)} = 0.5$. The suspiciousness scores of block 2, 4, and 5 are zeros since the numerator of Tarantula (i.e., $\frac{n_f^e}{n_f}$) is zero. The suspiciousness score of block 3 is: $\frac{1}{(\frac{2}{3}+1)} = 0.6$. Thus using Tarantula, the most suspicious block is block 3, followed by block 1, followed by blocks 2, 4, and 5. We could sort the basic blocks based on their suspiciousness scores and the debugger could check the blocks one-by-one from the most to the least suspicious block. Following Tarantula's recommendation, the fault could be found after one basic block inspection.

B. *Effectiveness Prediction*

The goal of our work is to predict if a particular fault localization tool is effective for a particular set of execution traces. We refer to the process where a fault localization tool is used to process a set of execution traces and output a list of suspicious program element as a *fault localization instance*. We define a fault localization instance to be effective if the root cause is located among the top-10 most suspicious program elements. Ties are randomly broken; this means that for example, if the top-20 program elements have the same suspiciousness scores, we randomly select 10 out of the 20 to be the top-10. Also, in case the root cause spans more than one program element (i.e., basic block) as long as one of the program elements is in the top-10, we consider the fault localization instance to be an effective one.

Various information could be leveraged to predict if a fault localization tool is effective given a set of program execution traces. We could investigate the execution traces. If there are very few failing execution traces, then it is likely to be harder for a spectrum based fault localization tool to differentiate faulty from correct program elements. In the extreme case, when there are no test cases that expose the fault (no failing execution traces), then the output of a fault localization tool cannot be effective. We could also investigate the output of the fault localization tool. In the special case where all program elements are given the same suspiciousness score, there is a very low likelihood that the fault localization tool will be effective for those execution traces.

III. Overall Framework

The goal of our framework is to build an oracle that is able to predict if a fault localization instance is effective or not. To realize this, our framework, illustrated in Figure 2, works on two phases: training and deployment. The training phase would output a model that is able to differentiate effective and ineffective fault localization instances. The deployment phase would apply this model to a number of unknown fault localization instances and output if the cases are likely to be effective or not. Let us describe these two phases in more detail.

In the training phase, we take in a set of fault localization instances. Some of these cases are effective and some others are ineffective. Each of these cases is represented by the following:

1) Program *spectra* corresponding to correct and faulty execution traces.
2) A list of *suspiciousness scores* that are assigned by the fault localization tools to the program elements.
3) An *effectiveness label*: effective (if the root cause is in the top-10) or ineffective (otherwise).

The training phase consists of two processes: feature extraction, and model learning. During feature extraction, based on a training data, we extract some feature values

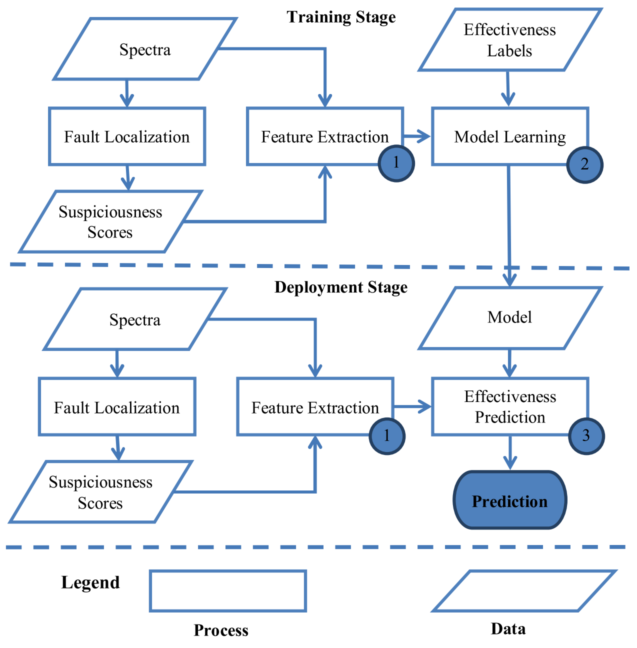

Figure 2. Proposed Framework

that shed light into some important characteristics that potentially differentiate effective from ineffective instances. In the model learning step, the feature values of each of the training instances along with the effectiveness labels are then used to build a discriminative model which is able to predict whether an unknown fault localization instance is effective or not. This discriminative model is output to the deployment stage.

The deployment stage consists of two blocks: feature extraction, and effectiveness prediction. We extract feature values from unknown instances whose labels, effective or ineffective, are to be predicted. These values are then fed to the discriminative model learned in the training phase. The model would then output a prediction.

We elaborate the feature extraction block in Section IV. The model learning and effectiveness prediction blocks are elaborated in Section V.

IV. Feature Extraction

We extract values of a number of features from input execution traces and from the outputs of a fault localization tool. Table II shows these features. We have in total 50 features. Fifteen of the features are extracted from input execution traces and the remaining thirty five features are extracted from the suspiciousness scores output by the tool.

The first fifteen input features capture information about program execution traces and program elements covered by these execution traces. Features T_1 to T_5 capture information on the number of execution traces available for fault localization. Too few number of traces might cause poor fault localization performance especially if there are too few failing traces. In the worst case where the number of failing traces is zero, the fault localization tool reduces to random guess. Features PE_1 to PE_4 capture the information on program elements that are covered by the execution traces. The more the number of program elements, the more difficulty a fault localization tool is likely to have as it needs to compare and differentiates more elements. With more program elements, the more likely a faulty program element to be assigned the same or lower suspiciousness scores as other program elements. Feature PE_5 captures cases where some program elements only appear in faulty but not correct executions. Intuitively, the chance for such cases to be effective is likely to be high. Feature PE_6 captures the opposite which might indicate omission errors: some program elements that should be executed are not executed. Features PE_7 to PE_{10} capture the two highest proportions of failures that passed by one program element. Intuitively, the higher the proportion of failures that passes a program element, the more likely it is the root cause.

The next thirty five output features capture the suspiciousness scores that are output by the fault localization tool. Features R_1 to R_{10} capture the top-10 suspiciousness scores. If the suspiciousness scores are too low, intuitively it is less likely for a fault localization instance to be effective. Features SS_1 to SS_6 compute some simple statistics of the top-10 suspiciousness scores. They serve as statistical summary of the scores. Features G_1 to G_{11} and C_1 to C_8 are aimed to capture a "break" or gap in the top-10 suspiciousness scores. This "break" shows that the localization tool is able to differentiate some program elements to be significantly more suspicious than the others. That might indicate that some of the top-10 program elements are probably to be the root cause. If the fault localization tool is unable to differentiate program elements, it is less likely to be effective. In the worst case, if it is unable to distinguish all program elements, fault localization again turns into random guess.

V. Model Learning & Effectiveness Prediction

We first describe our model learning process. Next, we describe how we apply the model to effectiveness prediction.

A. Model Learning

As inputs to this process, we have a set of training instances with their effectiveness labels. Each of the instance is represented as 50 feature values (aka. a feature vector) produced by the feature extraction process described in Section IV. The goal of the model learning process is to convert these set of feature vectors into a discriminative model that could predict the effectiveness label of a fault localization instance whose effectiveness is unknown.

We build upon and extend a state-of-the-art classification algorithm namely Support Vector Machine (SVM) [13]. SVM has been used in many past software engineering research studies, e.g., [2], [33], [25], [35], [36]. We first

Table II
LIST OF FEATURES (50 FEATURES)

ID	Description
Input: Traces (5 Features)	
T_1	Number of traces
T_2	Number of failing traces
T_3	Number of passing traces
T_4	$T_3 - T_2$
T_5	$\frac{T_2}{T_3}$
Input: Program Elements (10 Features)	
PE_1	Number of program elements covered in the failing execution traces
PE_2	Number of program elements covered in the correct execution traces
PE_3	$PE_2 - PE_1$
PE_4	$\frac{PE_1}{PE_2}$
PE_5	Number of program elements that appear only in failing execution traces
PE_6	Number of program elements that appear only in correct execution traces
PE_7	Highest proportion of failing execution traces that pass by one program element
PE_8	Second highest proportion of failing execution traces that pass by one program element
PE_9	$PE_7 - PE_8$
PE_{10}	$\frac{PE_8}{PE_7}$
Output: Raw Scores (10 Features)	
R_1	Highest suspiciousness score
R_2	Second highest suspiciousness score
R_i	i^{th} highest suspiciousness score, where $3 \leq i \leq 10$
Output: Simple Statistics (6 Features)	
SS_1	Number of distinct suspiciousness scores in $\{R_1, \ldots, R_{10}\}$
SS_2	Mean of $\{R_1, \ldots, R_{10}\}$
SS_3	Median of $\{R_1, \ldots, R_{10}\}$
SS_4	Mode of $\{R_1, \ldots, R_{10}\}$
SS_5	Variance of $\{R_1, \ldots, R_{10}\}$
SS_6	Standard deviation of $\{R_1, \ldots, R_{10}\}$
Output: Gaps (11 Features)	
G_1	$R_1 - R_2$
G_2	$R_2 - R_3$
G_i	$R_i - R_{(i+1)}$, where $3 \leq i \leq 9$
G_{10}	$Max_{1 \leq i \leq 9}(G_i)$
G_{11}	$Min_{1 \leq i \leq 9}(G_i)$
Output: Relative Differences (8 Features)	
C_1	$\frac{(R_2 - R_{10})}{(R_1 - R_{10})}$
C_i	$\frac{(R_{(i+1)} - R_{10})}{(R_1 - R_{10})}$, where $2 \leq i \leq 8$

describe standard off-the-shelf SVM. We then describe our extended SVM that handles the issue of imbalanced data caused since there are more ineffective fault localization instances than effective ones.

1) Off-the-Shelf SVM: SVM solves the classification problem by looking for a linear optimal separating hyperplane, which separates data instances of one class from another [37]. The chosen hyperplane is called *maximum marginal hyperplane* (MMH) in which the separation between two classes are maximized. For example, consider a training dataset in form of $(\vec{x_k}, y_k)$, where $\vec{x_k}$ is the feature vector of the k^{th} training data instance. Each y_k represents class label of data instance ($y_k \in \{+1, -1\}$). The problem of searching for a separating hyperplane with maximal margin could be reduced to finding the minimal value of $\frac{1}{2}\|\vec{w}\| = \frac{1}{2}\sqrt{{w_1}^2 + \cdots + {w_n}^2}$ which satisfies the constrains: $y_k(\vec{w} \cdot \vec{x_k} + b) \geq 1 \forall k$, where $\vec{w}$ is perpendicular to the separating hyperplan, n is the number of attributes, and b is a constant number indicates position of the hyperplan in multi-dimensional space. In this study, we use SVM^{light} version 6.02[1] with linear kernel.

2) SVM^{Ext}: Imbalanced training data is one of the issues that we encounter during the course of our study. There are more ineffective than effective fault localization instances. Thus we build upon standard off-the-shelf SVM to address this imbalanced data problem. We call our solution SVM^{Ext}.

The pseudo-code of our proposed SVM^{Ext} is shown in Figure 3. The algorithm takes as input a set of effective and ineffective fault localization instances - EI and II. We first check if there are more ineffective than effective localization instances (Line 1). If there are, we perform a data balancing step (Lines 2-8). We would like to duplicate effective instances that appear close to the hyperplane – these are effective instances that are close to one of the ineffective instances. In order to find these effective instances, we compute the similarity between each effective instance with each of the ineffective instances (Line 2). Each fault localization instance could be viewed as a 50-dimensional vector; each dimension is a feature and a localization instance is represented by the values of the 50 features described in Section IV. To measure the similarity between two instances we compute the Cosine similarity [29] of their representative vectors. Consider two vectors $(a_1, \ldots, a_{50})$ and $(b_1, \ldots, b_{50})$. The Cosine similarity of these two vectors is defined as:

$$\frac{\sum_{i=1}^{50}(a_i \times b_i)}{\sqrt{\sum_{i=1}^{50}(a_i)^2} \times \sqrt{\sum_{i=1}^{50}(b_i)^2}}$$

Next, for each effective instance, we calculate its highest similarity with an ineffective instance (Line 3). We sort the effective instances based on their highest similarities with ineffective instances (Line 4). We then insert these instances from the most similar to the least similar to the collection of effective instances EI until the number of effective instances matches that of ineffective ones (Lines 5-8). We then proceed to learn a model using off-the-shelf SVM and output the resultant model (Lines 9-10).

B. Effectiveness Prediction

The discriminative model learned in the model learning phase would be able to predict if an unknown instance (i.e., a fault localization instance whose effectiveness is unknown) is effective or not. The unknown instance needs to be transformed to a set of feature values using the feature extraction process described in Section IV. These feature values (aka. a feature vector) are then compared with the model and a prediction would be output. The feature vector is compared with the hyperplane that separates effective

[1]http://svmlight.joachims.org/

```
Procedure SVM$^{Ext}$
Inputs:
$EI$: Effective fault localization instances
$II$: Ineffective fault localization instances
Output: Discriminative model
Method:
1: If $(|EI| < |II|)$
2:     Let $S_i^j$ = Similarity between $EI[i]$ (i.e., the $i^{th}$ effective
           instance) with $II[j]$ (i.e., the $j^{th}$ ineffective instance)
3:     Let $M_i$ = $Max_{j \in \{0,\ldots,|II|-1\}} S_i^j$
4:     Let $MOSTSIM$ = Sorted $EI$ (sorted in descending
           order of $M_i$)
5:     Let $idx$ = 0
6:     While($|EI| < |II|$)
7:         Add $MOSTSIM[idx \% |MOSTSIM|]$ to $EI$
8:         $idx$++
9: Let $Model$ = Model learned with off-the-shelf SVM with
           $EI$ and $II$ as training data
10: Output $Model$
```

Figure 3. SVM^{Ext}

and ineffective training instances. Based on which side of the hyperplane the feature vector lies, the corresponding unknown instance is assigned either effective or ineffective prediction label.

VI. EXPERIMENTS

In this section we first describe our dataset, followed by our evaluation metrics, research questions, and results.

A. Dataset

We analyze 10 different programs. These include NanoXML, XML-Security, Space, and the 7 programs from the Siemens test suite [17]. These programs have been widely used in past studies on fault localization and thus could collectively be used as a benchmark [19], [27], [22], [1]. Table III provides the details on the programs.

NanoXML is an XML parsing utility written in Java. We download NanoXML from Software Infrastructure Repository (SIR) [8]. SIR contains 5 variants of NanoXML: NanoXML_v1, NanoXML_v2, NanoXML_v3, NanoXML_v4, and NanoXML_v5. Each of the variants contains faulty versions except NanoXML_v4. We downloaded all 32 faulty versions of these variants. We exclude two of the faulty versions as there are no failure-inducing test cases that expose the bugs. Thus, for NanoXML, in total, we analyze 30 faulty versions. XML-Security is a digital signature and encryption library written in Java. There are 3 variants of XML-Security in SIR: XMLSec_v1, XMLSec_v2, and XMLSec_v3. For each variant, several faulty versions are provided. In total, we downloaded 52 faulty versions from these variants; we analyze 16 of them, as there are no failure-inducing test cases that expose the other bugs. Space was used in European Space Agency and is an interpreter for Array Definition Language (ADL) written in C. All 35 faulty versions of Space downloaded from SIR are used for our experiments. For these 3 programs, in total we analyze, 81 faulty versions.

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

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

Siemens programs are originally created for a study on test coverage adequacy performed by researchers from Siemens Corporation Research [17]. Each of the seven programs has many faulty versions derived “by seeding realistic faults ” [17]. Each faulty version contains one bug that may span more than one program element (i.e., basic block). It comes with test cases and bug free versions. Siemens programs have been used in many fault localization studies including [19], [27], [22], [1]. The Siemens test suite[2] include the following programs: print_tokens, print_tokens2, replace, schedule, schedule2, tcas, and tot_info. There are a total of 132 versions in the test suite. We instrumented each blocks in the versions. We exclude versions that are seeded by bugs residing in variable declarations as our instrumentation cannot reach these declarations. Thus, we exclude the following versions: version 12 of replace, versions 13, 14, 15, 36, 38 of tcas, and versions 6, 10, 19, 21 of tot_info. Versions 4 and 6 of print_token are also excluded because they are identical with the bug free version. We exclude version 9 of schedule2 as running all test cases only produces correct executions – no test case is a failure-inducing one. In total, we include 119 faulty versions from Siemens test suite for our experiment. Adding the 81 faulty versions from the 3 other programs, we have in total 200 faulty versions.

B. Evaluation Metrics & Experiment Settings

We evaluate the accuracy of our solution in terms of precision, recall, and F-measure. These metrics have been frequently used to evaluate various prediction engines [13]. We first define the concepts of true positives, false positives, true negatives, and false negatives:

True Positives (TP):	Number of effective fault localization instances that are predicted correctly
False Positives (FP):	Number of ineffective fault localization instances that are predicted wrongly

[2]We use the variant at: www.cc.gatech.edu/aristotle/Tools/subjects

True Negatives (TN):	Number of ineffective fault localization instances that are predicted correctly
False Negatives (FN):	Number of effective fault localization instances that are predicted wrongly

Based on the above concepts, we can define precision, recall, and F-measure as follows:

$$Precision = \frac{TP}{TP + FP} \quad (1)$$

$$Recall = \frac{TP}{TP + FN} \quad (2)$$

$$F\text{-}Measure = \frac{2 \times Precision \times Recall}{Precision + Recall} \quad (3)$$

There is often a trade-off between precision and recall. Higher precision often results in lower recall (and vice versa). To capture whether an increase in precision (or recall) outweighs a reduction in recall (or precision), F-measure is often used. F-measure is the harmonic mean of precision and recall and it combines the two measures together into a single summary measure.

We perform ten-fold cross validation to evaluate the effectiveness of our proposed approach. Ten-fold cross validation is a standard approach in data mining to estimate the accuracy of a prediction engine [13]. Its goal is to assess how the result of a prediction engine generalizes to an independent test data. In ten-fold cross validation, we divide the dataset into ten groups. We use nine of the groups for training and one of the groups for testing. We repeat the process 10 times using different groups as the test group. We aggregate all the results and compute the final precision, recall, and F-measure.

C. Research Questions

We would like to answer the following research questions. The research questions capture different aspects that measure how good our proposed approach is.

RQ1. How effective is our approach in predicting the effectiveness of a state-of-the-art spectrum-based fault localization tool ?

We evaluate the accuracy of our tool in predicting the effectiveness of Tarantula which has been demonstrated to be one of the most accurate fault localization tools.

RQ2. How effective is our extended Support Vector Machine (SVM^{Ext}) compared with off-the-shelf Support Vector Machine (SVM) ?

To learn a discriminative model, we extend SVM to address the data imbalance issue. We would like to investigate if this extension is necessary to make our framework work.

RQ3. What are some important features that help in discriminating if a fault localization tool would be effective given a set of input traces ?

We investigate which of the 50 features that we use are more dominant and thus more effective to help us achieve higher prediction accuracy. In the machine learning community, Fisher score is often used to measure how dominant or discriminative a feature is [9], [11]. We compute the Fisher score of every feature as follows:

$$FS(j) = \frac{\sum_{class=1}^{\#class} (\bar{x}_j^{(class)} - \bar{x}_j)^2}{\sum_{class=1}^{\#class} (\frac{1}{n_{class}-1} \sum_{i=1}^{n_{class}} (x_{i,j}^{(class)} - \bar{x}_j^{(class)})^2)}$$

In the equation, $FS(j)$ denotes the Fisher score of the j^{th} feature. n_{class} is the numbers of data points (i.e., fault localization instances) with label $class$ (i.e., effective or ineffective). $\bar{x}_j$ denotes the average value of the j^{th} feature of all data points. $\bar{x}_j^{(class)}$ is the average value of the j^{th} feature of $class$-labeled data points. $x_{i,j}^{(class)}$ denotes the value of the j^{th} feature of the i^{th} $class$-labeled data point. Fisher score ranges from 0 to 1. A Fisher score of 0 indicates that a feature is not discriminative, while a Fisher score of 1 indicates that a feature is very discriminative.

RQ4. Could our approach be used to predict the effectiveness of different types of spectrum-based fault localization tool ?

There are different spectrum-based fault localization tools proposed in the literature. We would like to investigate if our approach also works for different spectrum-based fault localization tools. We consider two other well known spectrum-based fault localization tools: Ochiai [1], and Information Gain [24].

RQ5. How sensitive is our approach to the amount of training data ?

We use ten-fold cross validation to evaluate our approach. In ten-fold cross validation, we use 90% of the data for training, and the remaining 10% for testing. In this research question, we investigate the impact of reducing the number of training data on the accuracy of the proposed approach.

RQ6. Could data from one software program be used to train a discriminative model used to predict effectiveness of a fault localization tool on failures from other software programs ?

To answer this research question, we use data from N-1 (i.e., 9) software programs to build a model. This model is then used to predict the effectiveness of a fault localization tool on the remaining one software program. We refer to this process as N-fold cross-program validation.

D. Results

In this section, we answer our research questions one at a time by performing a set of experiments. For all research questions except RQ2, we use the default setting of our proposed framework presented in previous sections.

1) RQ1: Overall Accuracy: To answer our first research questions, we simply run Tarantula on the 200 faulty versions. We then predict if Tarantula is effective or not for each of the 200 faulty versions using SVM^{Ext}. We perform ten-fold cross validation and aggregate the result for the final precision, recall, and F-measure. For Tarantula, 85 of the localization instances are effective and 115 of the instances are ineffective. Thus, the data is imbalanced.

The result of our experiment is shown in Table IV. The result shows that we can achieve a precision of 54.36%. This means that we can correctly identify *many* ineffective fault localization instances (i.e., 47 out of the 115 ineffective instances). We can also achieve a recall of 95.29%. This means that we correctly identify *almost all* effective instances (i.e., 81 out of the 85 effective instances). F-measure, the harmonic mean of precision and recall, is often used to gauge on how effective a prediction engine is. Our F-measure is 69.23%. Comparing with many other studies performing other prediction tasks in software engineering research literature, e.g., [31], [32], our F-measure is comparable or higher.

Table IV
PRECISION, RECALL, AND F-MEASURE OF OUR PROPOSED APPROACH

Precision	54.36%
Recall	95.29%
F-Measure	69.23%

2) RQ2: SVM^{Ext} vs. SVM: Next, we compare our extended SVM (SVM^{Ext}) with standard off-the-shelf SVM. The precision, recall, and F-measure of using SVM^{Ext} and SVM is shown in Table V. SVM^{Ext} clearly outperforms SVM with respect to precision, recall, and F-measure. We also compute the relative improvement of SVM^{Ext} over SVM by the following formula:

$$Relative\ Improvement = \frac{(SVM^{Ext}\ Result - SVM\ Result)}{SVM\ Result}$$

We find that SVM^{Ext} outperforms SVM in terms of precision, recall, and F-Measure by 6.50%, 65.29%, and 27.87% respectively. SVM is not able to handle imbalanced data. The imbalanced data causes SVM to predict more unknown instances with the majority label that it sees in the training data (i.e., ineffective). This reduces the number of true positives and increases the number of false negatives, which causes a significant reduction in recall.

Table V
PRECISION, RECALL, AND F-MEASURE OF SVM^{Ext} AND SVM

	SVM^{Ext}	**SVM**	**Relative Improve.**
Precision	54.36%	51.04%	6.50%
Recall	95.29%	57.65%	65.29%
F-Measure	69.23%	54.14%	27.87%

3) RQ3: Important Features: Next, we investigate which features are important. We use Fisher score to rank the features. Table VI shows the list of top-10 most important features. Interestingly, we find that the top-10 features include input and output features. Both input execution traces and suspiciousness scores generated by a fault localization tool are important to predict the effectiveness of a fault localization instance.

Relative-difference features, i.e., C7, C8, C6, C5, and C1, are the most discriminative (5 out of the top-10 features). These features can capture a "break" or gap in the top-10 discriminative scores. This "break" signifies that the fault localization tool is able to differentiate some program elements to be significantly more suspicious than the others. Three of the top-10 features are related to program elements, i.e., PE1, PE2, and PE4. They capture the number of program elements covered in execution traces. The more program elements are covered, the harder it is to get effective fault localization as the fault localization tool needs to differentiate more program elements to find the root cause. The other two of the top-10 features are the highest suspiciousness score (R1) and the number of distinct suspiciousness scores in the top-10 scores (SS1). These are intuitively related to fault localization effectiveness: the higher a suspiciousness score is, the more likely a program element is the root cause; the more the number of distinct suspiciousness scores, the more that a fault localization tool differentiates program elements.

Table VI
TOP-10 MOST DISCRIMINATIVE FEATURES[2]

Rank	**Feature**	**Rank**	**Feature**
1	C7	6	SS1
2	C8	7	C5
3	C6	8	C1
4	PE1	9	PE4
5	PE2	10	R1

4) RQ4: Different Fault Localization Tools: We also investigate if our approach could be generalized to other spectrum-based fault localization tools aside from Tarantula. We use the same set of 200 faulty versions and perform the same ten-fold cross validation using SVM^{Ext} to evaluate two other spectrum-based fault localization tools: Ochiai [1], and Information Gain [24]. Table VII shows the precision, recall, and F-measure when we predict the effectiveness of Tarantula, Ochiai, and Information Gain.

We note that a similar precision, recall, and F-measure can be achieved for predicting the effectiveness of Ochiai and Information Gain. Our framework can achieve an F-measure of more than 75% for Ochiai and Information Gain. This is higher than the accuracy of our framework for Tarantula.

Table VII
PRECISION, RECALL, AND F-MEASURE FOR VARIOUS FAULT LOCALIZATION TOOLS

Tool	**Precision**	**Recall**	**F-Measure**
Tarantula	54.36%	95.29%	69.23%
Ochiai	63.23%	97.03%	76.56%
Information Gain	64.47%	93.33%	76.26%

[2]Please refer to Table II for the description of the features.

5) RQ5: Different Amount of Training Data: In ten-fold cross validation, we use 90% of the data for training on only 10% for testing. To answer this research question, we vary the amount of training data from 10% to 90% and show the resultant precision, recall, and F-measure. We randomly pick the data that we use for training. We show the result in Table VIII. Note that as we randomly resample the 90% data, the result is different with that of RQ1. We find that the performance of our framework does not degrade too much (F-measure $> 60\%$) if there is sufficient data for training (30-90%), the performance degrades significantly if there is too little training data (10-20%).

Table VIII
PRECISION, RECALL, AND F-MEASURE FOR VARIOUS AMOUNT OF TRAINING DATA

Amount of Data	Precision	Recall	F-Measure
90%	61.54%	100.00%	76.19%
80%	51.52%	100.00%	68.00%
70%	58.14%	100.00%	73.53%
60%	50.77%	97.06%	66.67%
50%	53.33%	95.24%	68.38%
40%	51.02%	98.04%	67.11%
30%	46.77%	98.31%	63.39%
20%	55.56%	36.76%	44.25%
10%	48.78%	26.32%	34.19%

6) RQ6: Cross-Program Setting: We perform N-fold cross-program validation to answer this research question. The result is shown in Table IX. The result shows that our approach could be used in cross-program setting with an F-measure of 63%, which is lower than our result for RQ1 (i.e., 69.23%). This is as expected as the programs are diverse and each program might have its own characteristics. It is thus harder to predict fault localization effectiveness for one program using training data from other programs.

Table IX
PRECISION, RECALL, AND F-MEASURE IN CROSS-PROGRAM SETTING

Precision	46.4%
Recall	100.00%
F-Measure	63.43%

E. Threats to Validity

We consider three kinds of threats to validity: internal, external, and constructing validity. Threats to internal validity corresponds to experimenter bias. In our experiments, we use the programs that are manually instrumented by Lucia et al. [24]. Due to the manual instrumentation process, there might be some basic blocks that are missed (i.e., no instrumentation code is added for them). Threats to external validity corresponds to the generalizability of our findings. In this study, we have analyzed 10 different programs. These programs are widely studied in past fault localization studies and thus collectively they can be used as a benchmark. We have also analyzed programs written in two programming languages: C and Java. Still, more programs can be analyzed to reduce the threat further. We plan to do this in a future work. Threats to construct validity corresponds to the suitability of our metrics. We use standard metrics of precision, recall, and F-measure. These are well known metrics in data mining, machine learning, and information retrieval and have been used in many past studies in software engineering, e.g., [16], [25], [2]. Thus with respect to these metrics, we believe there is little threat to construct validity. Another threat to construct validity is our definition of effective fault localization instance. In this preliminary study, we consider an instance is effective if at least one of the root cause is in the top-10 most suspicious program elements. Other definitions of effective fault localization could be considered, e.g., the root cause must be in the top-1 most suspicious program elements for an instance to be effective, etc. We leave the consideration of other definitions of effective fault localization for future work.

VII. RELATED WORK

In this section, we highlight a number of studies in *spectrum-based* fault localization which analyze program traces or their abstractions which capture the runtime behaviors of program.

Many spectrum-based fault localizations studies analyze two sets of program spectra: one set corresponding to correct executions, and another set corresponding to faulty executions [19], [1], [40], [21], [22], [30], [6], [23], [10], [20], [24], [3]. Based on these inputs, these studies would typically compute likelihood of different program elements to be the root cause of the faulty executions (aka. failures). Jones and Harrold propose Tarantula that computes the suspiciousness scores of various program elements by following this intuition: program elements that are executed more frequently by faulty executions rather than correct executions are deemed to be more suspicious [19]. Abreu et al. propose a different formula to compute suspiciousness scores [1]. They show that their proposed formula named Ochiai is able to outperform Tarantula. Zeller proposes Delta Debugging which compares a faulty execution and a correct execution and find the minimum state differences [40]. Liblit et al. compute predicates whose true evaluation correlates with failures [21]. This work is extended by Chao et al. which propose a work, named SOBER, that considers the repeated outcomes of predicate evaluations in a program run [22]. Santelices et al. use multiple program spectra to localize faults [30]. Cheng et al. propose an approach to mine a graph-based signatures, referred to as bug signatures, that differentiates correct from faulty executions [6]. Lo et al. extend the work of Cheng et al. by minimizing signatures and fusing minimized signatures to capture the context of program errors better [23]. Gong et al. after that propose a test case prioritization technique to reduce the number of test cases with known oracles for fault localization [10]. Gong et al. propose interactive fault localization where a fault localization tool iteratively updates its recommendation as it receives feedback from end users [20]. Lucia et al. investigate many association measures and adapt them for fault localization [24]. They find that information gain performs the best. Wang et al. employ search-based algorithms to combine various association measures and existing fault localization algorithms [38]. Artzi et al. use test generation for fault localization [3].

Other spectrum-based fault localizations analyze only one set of program spectra, i.e., faulty executions [41], [12], [18]. These techniques typically modify program runtime states systematically to localize faulty program elements. In this work, we focus on fault localization tools that compare correct and faulty executions.

VIII. Conclusion and Future Work

In this study, to address the unreliability of fault localization tool, we build an oracle that can predict the effectiveness of a fault localization tool on a set of execution traces. We propose 50 features that can capture interesting dimensions that potentially differentiate effective from ineffective fault localization instances. Values of these features from a training set of faulty localization instances can be used to build a discriminative model using machine learning. This model is then used to predict if unknown instances are effective or not. We have evaluated our solution on 200 faulty versions from NanoXML, XML-Security, Space, and the 7 programs in the Siemens test suite. Our solution can achieve a precision, recall, and F-measure of 54.36%, 95.29%, and 69.23%, respectively. We have also tested different aspects of our solution including its ability to handle cross-program setting and the results are promising.

As future work, we plan to improve the precision and F-measure of our proposed approach further. We plan to perform an in-depth analysis of cases where our proposed approach is less effective and design appropriate extension to the approach. We would also like to extend our approach to predict the effectiveness of other fault localization techniques, e.g., [6], [38], [30], [7]. We also plan to investigate the effectiveness of and incorporate some findings from recent studies on learning from imbalanced data performed in the data mining community [15] to further improve our SVM^{Ext}. It is also interesting to leverage other information aside from execution traces; some failures come with textual descriptions [42], and it would be interesting to employ advanced text mining solutions [5], [39] to identify whether fault localization tools would be effective on such failures.

References

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

[2] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *ICSE*, 2006, pp. 361–370.

[3] S. Artzi, J. Dolby, F. Tip, and M. Pistoia, "Directed test generation for effective fault localization," in *ISSTA*, 2010.

[4] B. Beizer, *Software Testing Techniques*, 2nd ed. Boston: International Thomson Computer Press, 1990.

[5] D. Blei, A. Ng, and M. Jordan, "Latent Dirichlet allocation," *J. Machine Learning Research*, vol. 3, pp. 993–1022, 2003.

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

[7] T. Chilimbi, B. Liblit, K. Mehra, A. Nori, and K. Vaswani, "HOLMES: Effective statistical debugging via efficient path profiling," in *ICSE*, 2009.

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

[9] R. Duda, P. Hart, and D. Stork, *Pattern Classification*. Wiley-Interscience Publication, 2001.

[10] L. Gong, D. Lo, L. Jiang, and H. Zhang, "Diversity maximization speedup for fault localization," in *ASE*, 2012, pp. 30–39.

[11] Q. Gu, Z. Li, and J. Han, "Generalized fisher score for feature selection," in *UAI*, 2011, pp. 266–273.

[12] N. Gupta, H. He, X. Zhang, and R. Gupta, "Locating faulty code using failure-inducing chops," in *ASE*, 2005, pp. 263–272.

[13] J. Han and M. Kamber, *Data Mining Concepts and Techniques*, 2nd ed. Morgan Kaufmann, 2006.

[14] M. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi, "An empirical investigation of the relationship between spectra differences and regression faults." *Software Testing, Verification and Reliability*, 2000.

[15] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Trans. Knowl. Data Eng.*, vol. 21, no. 9, pp. 1263–1284, 2009.

[16] L. Huang, V. Ng, I. Persing, R. Geng, X. Bai, and J. Tian, "Autoodc: Automated generation of orthogonal defect classifications," in *ASE*, 2011.

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

[18] D. Jeffrey, N. Gupta, and R. Gupta, "Fault localization using value replacement," in *ISSTA*, 2008.

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

[20] Liang Gong, David Lo, Lingxiao Jiang, and Hongyu Zhang, "Interactive fault localization leveraging simple user feedback," in *ICSM*, 2012, pp. 67–76.

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

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

[23] D. Lo, H. Cheng, and X. Wang, "Bug signature minimization and fusion," in *HASE*, 2011, pp. 340–347.

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

[25] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y.-G. Guéhéneuc, G. Antoniol, and E. Aïmeur, "Support vector machines for anti-pattern detection," in *ASE*, 2012.

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

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

[28] T. Reps, T. Ball, M. Das, and J. Larus, "The use of program profiling for software maintenance with applications to the year 2000 problem." in *ESEC/FSE*, 1997.

[29] G. Salton and M. McGill, *Introduction to Modern Information Retrieval*. McGraw-Hill, 1983.

[30] R. Santelices, J. Jones, Y. Yu, and M. Harrold, "Lightweight fault-localization using multiple coverage types," in *ICSE*, 2009.

[31] H. Seo and S. Kim, "Predicting recurring crash stacks," in *ASE*, 2012, pp. 180–189.

[32] E. Shihab, A. Ihara, Y. Kamei, W. Ibrahim, M. Ohira, B. Adams, A. E. Hassan, and K. Matsumoto, "Studying re-opened bugs in open source software," *Empirical Software Engineering*, 2012.

[33] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *ICSE (1)*, 2010.

[34] G. Tassey, "The economic impacts of inadequate infrastructure for software testing." *National Institute of Standards and Technology. Planning Report 02-3.2002*, 2002.

[35] F. Thung, D. Lo, and L. Jiang, "Automatic defect categorization," in *WCRE*, 2012.

[36] Y. Tian, C. Sun, and D. Lo, "Improved duplicate bug report identification," in *CSMR*, 2012, pp. 385–390.

[37] V. Vapnik, *The Nature of Statistical Learning Theory*, 2nd ed. Springer-Verlag, 2000.

[38] S. Wang, D. Lo, and L. Jiang, "Search-based fault localization," in *ASE*, 2011.

[39] X. Wang, D. Lo, J. Jiang, L. Zhang, and H. Mei, "Extracting paraphrases of technical terms from noisy parallel software corpora," in *ACL/IJCNLP*, 2009.

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

[41] X. Zhang, N. Gupta, and R. Gupta, "Locating faults through automated predicate switching," in *ICSE*, 2006.

[42] 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 *ICSE*, 2012.