2017 IEEE 25th International Conference on Program Comprehension (ICPC)

How Does Execution Information Help with Information-Retrieval Based Bug Localization?

Tung Dao
Computer Science
Virginia Tech
Blacksburg, VA 24060
tungdm@vt.edu

Lingming Zhang
Computer Science
The University of Texas at Dallas
Dallas, TX 75080
lingming.zhang@utdallas.edu

Na Meng
Computer Science
Virginia Tech
Blacksburg, VA 24060
nm8247@cs.vt.edu

***Abstract*—Bug localization is challenging and time-consuming. Given a bug report, a developer may spend tremendous time comprehending the bug description together with code in order to locate bugs. To facilitate bug report comprehension, information retrieval (IR)-based bug localization techniques have been proposed to automatically search for and rank potential buggy code elements (i.e., classes or methods). However, these techniques do not leverage any dynamic execution information of buggy programs. In this paper, we perform the first *systematic* study on how *dynamic* execution information can help with *static* IR-based bug localization. More specifically, with the fixing patches and bug reports of 157 real bugs, we investigated the impact of various execution information (i.e. coverage, slicing, and spectrum) on three IR-based techniques: the baseline technique, BugLocator, and BLUiR.**

Our experiments demonstrate that both the coverage and slicing information of failed tests can effectively reduce the search space and improve IR-based techniques at both class and method levels. Using additional spectrum information can further improve bug localization at the method but not the class level. Some of our investigated ways of augmenting IR-based bug localization with execution information even outperform a state-of-the-art technique, which merges spectrum with an IR-based technique in a complicated way. Different from prior work, by investigating various easy-to-understand ways to combine execution information with IR-based techniques, this study shows for the *first* time that execution information can *generally* bring *considerable* improvement to IR-based bug localization.

I. Introduction

Bug localization, or fault localization, is important in software maintenance, because effective bug fixing relies on precise bug location information. However, given a bug report and a buggy program, developers may spend tremendous time and effort understanding the bug description and code to locate bugs. To facilitate bug comprehension and accelerate bug finding, researchers have proposed various IR-based bug localization techniques [36], [21], [19], [20]. By treating the bug report as a query, and the source code files as plain documents, these techniques rank software entities (i.e., classes or methods) based on their relevance or similarity to the query. The more relevant a program entity is, the higher it is ranked as a potential bug location.

These IR-based techniques can facilitate bug localization and program comprehension, because they help developers focus effort on bug-relevant code elements. In particular, Zhou et al. proposed BugLocator to use a specialized Vector Space Model (VSM), called rVSM, by considering file lengths and bug history [36]. They demonstrated that rVSM outperformed other IR models on real bugs from four open-source projects. Saha et al. further proposed BLUiR [21] to use another revised VSM by considering code constructs, such as class and method names. Their experiments showed that BLUiR even outperformed BugLocator.

Despite the various IR-based techniques, we are curious whether the execution information of buggy programs can further help bug localization and program comprehension. Le et al. proposed the first tool, AML, to combine IR-based bug localization with spectrum execution information [13]. They used a hybrid model to encode both spectrum and textual information into a specialized VSM. They found that AML outperforms Learning-to-rank [32] (a state-of-the-art IR-based bug localization technique), and MULTRIC [30] (a state-of-the-art spectrum-based bug localization technique). However, it is still unknown *how various types of execution information can generally help with IR-based bug localization*.

To systematically investigate the impact of various execution information on IR-based techniques, we performed an extensive study on three kinds of execution information, and three state-of-the-art IR-based techniques, using an existing dataset of 157 real bugs. More specifically, we investigated the following three types of information: (1) **coverage**—the classes or methods covered by failed tests, (2) **slicing**—the classes or methods in the dynamic slice [29] of each failure witness statement (i.e., a failure assertion or an exception-throwing statement), and (3) **spectrum**—the suspiciousness score of each executed class or method, which describes the coverage ratio between passed and failed tests [10]. Hypothetically, coverage and slicing may help with IR-based techniques by refining the search space. The reason is if an entity (i.e., class or method) is not covered by a failed test or does not occur in the slice of a failure witness statement, it is unlikely to be buggy. Spectrum information may further help by ranking program entities purely based on suspiciousness scores. Its ranking can complement the ranking by IR-based techniques.

In this study, we experimented with three existing IR-based techniques: the baseline, BugLocator [36], and BLUiR [21]. To assess the impact of different execution information on

978-1-5386-0535-6/17 $31.00 © 2017 IEEE
DOI 10.1109/ICPC.2017.29

IR-based techniques, we combined IR-based techniques and execution information in four ways: (1) $\mathbf{IR}_c$—to combine coverage with IR, (2) $\mathbf{IR}_s$—to combine slicing with IR, (3) $\mathbf{IR}_{cp}$—to combine coverage and spectrum with IR, and (4) $\mathbf{IR}_{sp}$—to combine slicing and spectrum with IR.

Our experiments revealed a number of interesting findings. First, we observed that coverage information can effectively reduce the search space of IR-based techniques, and thus significantly improve bug localization at both class and method levels. In particular, for all three IR-based techniques, the number of actual bug locations ranked within Top 10 was increased by 17-33% at class level, and by 62-100% at method level. This combination strategy even outperformed state-of-the-art hybrid technique AML [13] in most cases. Second, slicing information can further improve bug localization. Compared with coverage, slicing further increased the number of actual bug locations among Top 10 by 1-43% at class level, and by 9-30% at method level. Third, the additional usage of spectrum information further improved bug localization at method level. Our study shows that dynamic execution information can *generally* bring *considerable* improvement to IR-based bug localization. Some future approaches that delicately combine various execution information with IR-based techniques may further facilitate bug localization and program comprehension.

In summary, this paper makes the following contributions:

- We investigated four ways to combine execution information with IR-based bug localization by exploring three kinds of information, and three IR-based techniques.
- Our quantitative analysis shows that coverage and slicing information effectively helps with IR-based bug localization at both the class and method levels, while spectrum information further helps at method level.
- This empirical study shows for the first time that execution information can generally bring considerable improvement to IR-based bug localization, even when the combination strategies are simple and easy to understand.

II. Methodology

In this section, we first present the background of IR-based bug localization (Section II-A), and then discuss different execution information and how we collected them (Section II-B). Finally, we explain our four ways of combining execution information with IR-based techniques (Section II-C).

A. IR-Based Bug Localization

Given a bug report, IR-based bug localization [15], [27], [22], [16] treats the report as a *query*, and considers source code elements as a *document collection*. It ranks elements according to their textual similarity with the report. There are several approaches proposed [36], [28], [19], [12], [23], [15], [21]. In this section, we summarize the baseline IR-based technique, two widely used IR-based tools (BugLocator [36] and BLUiR [21]), and AML—a hybrid approach combining an IR-based technique with spectrum execution information.

The baseline technique applies the IR framework Indri [24] directly without any optimization. Given a bug report and a buggy program, it preprocesses the data in three steps. First, it extracts all words except for stop words (e.g., "a", "at", and "which"), and programming language keywords (e.g., `while`, `for`). Second, it applies camel case splitting (`IsSigned` $\rightarrow$ {"Is", "Signed"}) and stemming [8] ("Signed" $\rightarrow$ "sign") to split and stem code identifiers. Third, it indexes all documents (i.e., classes or methods) by terms, and computes the term frequency (TF) for individual documents. After the preprocessing, Indri takes in the bug report query and document corpus, retrieves query-relevant documents, and ranks the documents by relevance. For our study, we used the default VSM (i.e., TF-IDF) model of Indri to do experiments.

BugLocator [36] improves the baseline with two specializations. First, instead of using the default VSM, BugLocator builds a revised VSM (rVSM) to calculate the query-document similarity differently. The specialization is based on the tool builders' observation that longer files are more likely to be buggy than shorter ones. Second, when ranking documents, BugLocator also considers bug history. Hypothetically, similar bug reports may indicate similar bug locations. If a new report is similar to some reports whose bugs are already located, then BugLocator highly ranks those bug locations. According to the evaluation with more than 3000 real bugs in prior work [36], these two customizations enabled BugLocator to outperform the known baseline techniques [20], [17].

BLUiR [21] improves the baseline by considering structure information. Different from prior approaches, BLUiR observes document structures and program structures. Given a bug report, it assigns more weight to terms in titles than those in summaries, because report titles usually provide more relevant information. Given a program, BLUiR assigns more weight to names of classes and methods, but less to variable names and comments, because it assumes class and method names are more important. Prior work [21] showed that BLUiR outperformed BugLocator and BugScout [19].

AML [13] is a hybrid approach to combine spectrum execution information (see Section II-B) with an IR-based technique. It consists of three components: AML^{Text} (the IR-based tool), $\text{AML}^{\text{Spectra}}$ (the spectrum information), and $\text{AML}^{\text{SuspWord}}$. These components independently calculate suspiciousness scores of every program element, and AML then computes a weighted sum of the scores to rank program elements. Although AML outperforms the state-of-the-art IR-based technique, it is still unclear *whether the outperformance is due to the approach design or extra dynamic information*.

B. Execution Information and Its Collection

In this section, we overview the three most widely used types of execution information: coverage, slicing, and spectrum. We also explain why they may help bug localization, and how we collected them.

Coverage information and its collection. Coverage describes all entities (i.e., classes or methods) covered by a program execution. This information can help bug localization because if a test fails, the failure run should cover some buggy entities. To collect the information, we used ASM bytecode

manipulation framework [1] to instrument the entry and exit of each method. This allows us to record which methods are executed at runtime, and to identify the executed classes that own the executed methods. We used Java Agent to insert code instrumentation on-the-fly during class loading time.

Slicing information and its collection. Slicing [29] describes all classes or methods that may affect the state of a program point. This information can be helpful because when a test fails, only statements responsible for the failure can be buggy. In other words, given a failure witness statement (i.e., the failed assertion or the statement that threw an uncaught exception), only statements on which it is transitively control or data dependent are responsible for the failure.

Slicing information can be collected statically or dynamically [29], [25], [6]. We used JavaSlicer [2], a dynamic slicing tool, to instrument every *instruction* for trace collection, and to perform backward slicing from the failure witness statement in each trace [25]. We chose the tool for two reasons. First, we prefer dynamic slicing to static slicing, because dynamic slicing identifies all code elements that *actually* affect the failure state. Second, unlike other dynamic slicing tools, JavaSlicer is publicly available and widely used [31], [35]. Although it outputs all instructions responsible for a failure state, in our study, we mapped them to their owner methods or classes for method-level or class-level slices, because IR-based bug localization ranks buggy methods or classes. Specifically, if one method or class has at least one instruction in the failure-relevant slice, we include the entity into the method-level or class-level slice.

Spectrum information and its collection. Spectrum describes how suspicious a program element is when the program fails. The higher suspiciousness score a code element gets, the more likely it is buggy. Intuitively, if a code element is executed solely by failed tests but never by passed tests, the element may be buggy. Spectrum information can help IR-based bug localization, because it provides a complementary approach to localize bugs.

In our study, with the ASM bytecode instrumentation mentioned above, we got both method-level and class-level coverage information by passed tests and failed tests. Then we tried four widely-used formulae to separately compute the spectrum information: Tarantula [10], Ochiai [3], Jaccard [4], and Ample [5]. Formally, given a buggy program and a set of tests, we use n_f and n_p to represent the total number of failed and passed tests. For each program element e, whether it is a class or a method, we use e_f and e_p to denote the number of failed and passed tests executing e. All four formulae are shown below:

$$Tarantula = \frac{\frac{e_f}{e_f+n_f}}{\frac{e_f}{e_f+n_f}+\frac{e_p}{e_p+n_p}} \quad (1)$$

$$Ochiai = \frac{e_f}{\sqrt{(e_f+e_p)(e_f+n_f)}} \quad (2)$$

$$Jaccard = \frac{e_f}{e_f+e_p+n_f} \quad (3)$$

$$Ample = |\frac{e_f}{e_f+n_f} - \frac{e_p}{e_p+n_p}| \quad (4)$$

C. Combining Execution Information with IR-Based Bug Localization Techniques

Given a bug report, a buggy program, the program's passed tests and failed tests, we aim to improve IR-based bug localization with execution information using two heuristics: search space reduction and rank tuning. We systematically investigated four approaches to combine the two types of information. Intuitively, the combination approaches should always lead to considerable improvement, since both static and dynamic information is used. However, we do not know how effectively execution information can help improve IR-based techniques.

a) Search space reduction: According to the Propagation, Infection, and Execution (PIE) model [26], bugs are triggered when a buggy element is executed. Given a buggy program and failed tests, we use coverage information of the failed tests to reduce the search space of IR-based techniques. If an element is not covered by any failure run, it is always irrelevant to failures, and gets excluded from the scope.

Similarly, slicing information can also be used to refine the search space of IR-based techniques, because only elements affecting the runtime state of a fault witness statement may be buggy. If an element is not in any failure-relevant slice, it is unrelated to the reported bug.

b) Rank tuning: Given a ranked list by an IR-based technique, coverage or slicing information always shortens the list, but does not change the relative ranking among covered or sliced elements. If two failure-relevant elements `A` and `B` are ranked in a wrong order, neither coverage nor slicing can correct the mistake. In comparison, spectrum information maps each element to a suspiciousness score based on execution coverage. According to the suspiciousness scores, spectrum may rank code elements very differently from IR-based techniques. When combining the two ranked lists together, we may correct the ordering mistake mentioned above in an IR-based list. Formally, given a class or method whose source code is represented as d, if its IR-based score is denoted as $Score(d,q)$, and spectrum-based score is $Susp(d)$, we define a combination factor α to control their separate weights when synthesizing an adjusted score $Score'(d,q)$ as follows:

$$Score'(d,q) = (1-\alpha) * Score(d,q) + \alpha * Susp(d) \quad (5)$$

where α is configured to vary from 0 to 1, with 0.1 increment. In this way, we are able to experiment with different configurations to identify the optimal combination.

c) Four variants: We experimented four ways of combining dynamic execution information with IR-based techniques.

IR$_c$: Coverage information is used to refine the search space of IR-based techniques by filtering out unexecuted entities (i.e., classes or methods). Ideally, the filtering can

be applied either before or after IR-based techniques, namely (1) *Filter-then-IR* or (2) *IR-then-Filter*. If filtering is applied first, IR-based techniques only focus on documents covered by failed tests. Otherwise, IR-based techniques are applied to the whole codebase, and then coverage is used to remove entities from the ranked lists of IR-based techniques. Intuitively, both approaches should work equally well. However, according to our experiments (Section V-A), approach (2) is generally better, so we used *IR-then-Filter* by default.

IR$_s$: Similar to IR$_c$, slicing information is used to refine the search space of IR-based bug localization.

IR$_{cp}$: Coverage information is first used to refine an IR-based list. Spectrum information is then applied to synthesize a tuned ranked list.

IR$_{sp}$: Slicing information is first used to refine an IR-based list. Spectrum information is then used to tune the ranked list.

To systematically compare different combination approaches, we evaluated their effectiveness at both class and method levels. For class-level evaluation, we check whether an approach localizes the buggy class(es). For method-level evaluation, we verify whether an approach identifies the buggy method(s). Note that since IR-based bug localization suggests buggy classes and methods, our investigated combinations also rank class- or method-level bug locations.

III. Research Questions

In this empirical study, we aim to answer the following research questions:

Research Question 1: *How does coverage information help with IR-based bug localization?*

Intuitively, by refining search space, coverage information should help. However, it is unclear how effectively coverage information achieves improvement.

Research Question 2: *How does slicing information help with IR-based bug localization?*

We are curious how slicing information helps with IR-based bug localization by reducing the search space.

Research Question 3: *How does spectrum information further improve bug localization over IR$_c$ and IR$_s$?*

Hypothetically, by integrating spectrum with IR$_c$ and IR$_s$, we should localize bugs more effectively. The reason is coverage and slice only focus on the execution of failed tests, but spectrum also takes passed tests into consideration. With more execution information included, we may achieve improvement in bug localization effectiveness. However, it is unclear how much improvement we can get.

Research Question 4: *How do our simple combinations compare with AML?*

We are curious how well our combination approaches work in comparison with the state-of-the-art hybrid technique. If our approaches work equally well or even better, it means that dynamic information generally helps IR-based techniques, no matter how simply the combination is done.

TABLE I: Dataset

Project	#Bug	Class		Method	
		#Total	#Buggy	#Total	#Buggy
AspectJ	41	4,157	67	14,218	88
Ant	53	1,063	96	9,624	197
Lucene	37	2,737	158	10,220	311
Rhino	26	191	58	4,839	145
Overall	**157**	**8,148**	**379**	**38,901**	**741**

IV. Experiment Settings

We experimented with the existing benchmark suite published by Le et al. [13]. As shown in Table I, the dataset consists of 157 real bugs extracted from 4 open source Java projects: AspectJ, Ant, Lucene, and Rhino. For each bug, the dataset includes a bug report, a set of test cases including passed and failed tests, a buggy program, and a fixed version of the program. The bug report is used by IR-based techniques to locate bugs. The test cases are used for execution information collection. The actual bug fix, which is the textual *diff* between the buggy program and its revised version, serves as the ground truth to evaluate whether a bug is located correctly. As a bug fix may involve changes to a single or multiple classes or methods, if we consider all modified code elements as bug locations, we have 157 bugs mapped to 379 buggy classes, or 741 buggy methods.

We used the following three widely used metrics [36], [21] to measure the effectiveness of bug localization techniques:

Recall at Top N counts the number of actual buggy entities included in the top N (= 1, 5, 10) ranked results. Given the same N, the more buggy entities are included, the better a bug localization approach works.

Mean Average Precision (MAP) calculates the average precision values among a set of queries. The higher value, the better. The Average Precision (AP) of a single query is defined as:

$$AP = \sum_{k=1}^{M} \frac{P(k) * pos(k)}{\text{number of positive instances}} \tag{6}$$

Suppose given a query (e.g., a bug report), M documents are retrieved and only one of them is positive (i.e., buggy). Then in the formula, the number of positive instances is equal to 1. k varies from 1 to M. For each value of k, $P(k)$ is the percentage of positive documents among the top k documents, and $pos(k)$ is a binary indicator of whether or not the k^{th} document is positive. For example, if 5 documents are retrieved, and the 4^{th} and 5^{th} are positive, then AP is $(\frac{1}{4} + \frac{2}{5})/2 = 0.325$.

Mean Reciprocal Rank (MRR) measures precision in a different way. Given a set of queries, it calculates the mean of reciprocal rank values for all queries. The higher value, the better. The Reciprocal Rank (RR) of a single query is defined as:

$$RR = \frac{1}{rank_{best}} \tag{7}$$

where $rank_{best}$ is the rank of the first relevant document found. For example, for a given query, if 5 documents are retrieved, and the 4^{th} and 5^{th} are relevant, then RR is $\frac{1}{4} = 0.25$.

TABLE II: F-I vs. I-F at class level

Metric	Project	Baseline		BugLocator		BLUiR	
		F-I	I-F	F-I	I-F	F-I	I-F
Top 1	AspectJ	7	6	4	6	4	5
	Ant	29	32	31	33	27	31
	Lucene	14	14	12	15	11	14
	Rhino	3	4	3	9	8	11
	Overall	**53**	**56**	**50**	**63**	**50**	**61**
Top 5	AspectJ	18	18	16	16	17	19
	Ant	54	59	54	55	59	60
	Lucene	45	47	50	48	53	51
	Rhino	16	18	20	19	19	20
	Overall	**133**	**142**	**140**	**138**	**148**	**150**
Top 10	AspectJ	27	25	21	25	25	28
	Ant	63	64	62	63	65	68
	Lucene	61	59	67	59	72	68
	Rhino	25	26	25	28	26	25
	Overall	**176**	**174**	**175**	**175**	**188**	**189**
MAP	AspectJ	0.25	0.23	0.22	0.25	0.22	0.25
	Ant	0.63	0.73	0.70	0.71	0.67	0.72
	Lucene	0.50	0.49	0.49	0.54	0.50	0.55
	Rhino	0.33	0.40	0.34	0.47	0.50	0.55
	Mean	**0.43**	**0.46**	**0.44**	**0.49**	**0.47**	**0.52**
MRR	AspectJ	0.28	0.27	0.24	0.29	0.24	0.29
	Ant	0.67	0.75	0.73	0.76	0.69	0.75
	Lucene	0.60	0.59	0.55	0.61	0.58	0.64
	Rhino	0.32	0.40	0.36	0.51	0.51	0.56
	Mean	**0.47**	**0.50**	**0.47**	**0.54**	**0.51**	**0.56**

V. Results and Analysis

In this section, we first show how effectively coverage and slicing can improve IR-based techniques (Section V-A and V-B). Then we describe the effectiveness of spectrum (Section V-C). Finally, we compare our combination approaches with AML (Section V-D).

A. RQ1: How does coverage information help with IR-based bug localization?

We compared IR_c with the original IR-based techniques at both class and method levels. Since coverage can be used to refine the search space either *before* or *after* IR-based techniques, we first investigated which order always produces better results.

Filter-then-IR (F-I) vs. IR-then-Filter (I-F). The former one first uses coverage information to scope a list of entities (i.e. classes or methods) executed by failed tests, and then applies IR-based techniques to rank entities relevant to a given bug report. The latter one takes the two steps in a reverse order. To understand which option is better, we tried both options to localize bugs at class and method levels, and observed that I-F performed better in most cases. Due to the space limit, we only show the class-level results in Table II. One possible reason is that I-F leverages the whole codebase to build corpus for IR techniques, while F-I only uses the executed classes or methods. With a larger document corpus, I-F better identifies both important and unimportant words, and thus ranks executed documents more precisely. Therefore, by default, we used I-F to integrate coverage or slicing with IR-based techniques.

> **Finding 1:** *Compared with Filter-then-IR, IR-then-Filter worked better to refine the search space of IR-based bug localization with execution information.*

Class-level bug localization identifies buggy classes. Table III (a) shows the comparison between IR-only and IR_c for class-level bug localization. Under each IR-based technique (Baseline, BugLocator, or BLUiR), there are two columns: **IR** and **IR_c**. Each column "IR" shows the original technique's results, while column "IR_c" presents the results of the hybrid approach. Surprisingly, coverage information alone greatly boosted the overall effectiveness for all IR-based techniques. In particular, the MAP value of BugLocator was significantly improved from 0.28 to 0.49, while the MRR value was improved from 0.34 to 0.54. Among the three techniques, BLUiR had the best effectiveness, which conformed with the findings in prior work [21]. When augmented with coverage information, BLUiR outperformed others for all metrics except for Top 1.

However, the effectiveness improvement by coverage did not evenly distribute among different projects. For example, compared with Baseline, IR_c improved the Top-1 metric of AspectJ from 4 to 6 with 50% improvement, but did not improve the metric for Rhino. We examined Rhino's source code, and found that the actual buggy classes were usually ranked very low (e.g., below Top 100). Therefore, even though coverage could effectively shorten ranked lists, it was not capable of removing hundreds of unexecuted classes to promote any buggy class to Top 1.

> **Finding 2:** *At class level, coverage consistently improved all studied IR-based techniques. On average, MAP was increased from 0.34 to 0.49 with 44% increment, and MRR was increased from 0.40 to 0.53 with 33% increment.*

Method-level bug localization isolates buggy methods for developers to examine. Compared with class-level bug localization, this approach can save more manual effort, because it does not leave a whole class body for developers to delve into [13]. Table III (b) presents the results. Compared with class level, all three original techniques worked more poorly at method level, meaning that locating buggy methods is generally harder than locating classes. Two reasons can explain the difficulty. First, each method contains fewer terms to index, and may become less relevant to random queries. Second, there are many more methods to rank than classes, which makes it harder to rank the actual buggy methods high.

Compared with class level, the improvement by coverage was more significant at method level. For BLUiR, the overall MAP and MRR improvements were 114% (from 0.14 to 0.30) and 84% (from 0.19 to 0.35), while the class-level improvements in Table III (a) were 30% (from 0.40 to 0.52) and 24% (from 0.45 to 0.56). Across all subjects, coverage effectively improved IR-based techniques in most cases.

As shown in Table III (b), among different techniques, BLUiR performed the worst without coverage information. This observation complements the findings in prior work [21], because Saha et al. only evaluated BLUiR's performance at class level. The reason why the observations at class level

TABLE III: IR vs. IR_c bug localization

Metric	Project	(a) Class Level						(b) Method Level					
		Baseline		BugLocator		BLUiR		Baseline		BugLocator		BLUiR	
		IR	IR_c	IR	IR_c	IR	IR_c	IR	IR_c	IR	IR_c	IR	IR_c
Top 1	AspectJ	4	6	2	6	3	5	3	3	2	4	2	3
	Ant	27	32	22	33	26	31	9	12	10	13	6	13
	Lucene	12	14	6	15	11	14	4	7	3	9	5	7
	Rhino	4	4	5	9	11	11	4	6	4	6	5	6
	Overall	**47**	**56**	**35**	**63**	**51**	**61**	**20**	**28**	**19**	**32**	**18**	**29**
Top 5	AspectJ	13	18	7	16	12	19	4	8	3	8	4	7
	Ant	48	59	44	55	45	60	21	33	20	36	16	36
	Lucene	34	47	32	48	41	51	14	30	15	33	16	32
	Rhino	15	18	13	19	19	20	7	10	8	14	7	15
	Overall	**110**	**142**	**96**	**138**	**117**	**150**	**46**	**81**	**46**	**91**	**43**	**90**
Top 10	AspectJ	18	25	14	25	20	28	5	12	6	16	6	12
	Ant	55	64	49	63	59	68	30	47	29	51	25	54
	Lucene	53	59	51	59	59	68	24	37	30	44	24	42
	Rhino	21	26	18	28	23	25	9	14	10	21	10	22
	Overall	**147**	**174**	**132**	**175**	**161**	**189**	**68**	**110**	**75**	**129**	**65**	**130**
MAP	AspectJ	0.15	0.23	0.10	0.25	0.14	0.25	0.08	0.11	0.07	0.14	0.06	0.10
	Ant	0.55	0.73	0.50	0.71	0.54	0.72	0.17	0.34	0.23	0.37	0.15	0.36
	Lucene	0.34	0.49	0.26	0.54	0.39	0.55	0.13	0.36	0.09	0.32	0.15	0.38
	Rhino	0.34	0.40	0.28	0.47	0.51	0.55	0.17	0.29	0.18	0.32	0.19	0.34
	Mean	**0.35**	**0.46**	**0.29**	**0.49**	**0.40**	**0.52**	**0.14**	**0.28**	**0.14**	**0.29**	**0.14**	**0.30**
MRR	AspectJ	0.18	0.27	0.12	0.29	0.17	0.29	0.09	0.12	0.07	0.16	0.07	0.12
	Ant	0.61	0.75	0.55	0.76	0.59	0.75	0.24	0.40	0.27	0.43	0.19	0.41
	Lucene	0.49	0.59	0.35	0.61	0.50	0.64	0.23	0.44	0.20	0.46	0.28	0.48
	Rhino	0.35	0.40	0.32	0.51	0.54	0.56	0.20	0.34	0.20	0.36	0.23	0.37
	Mean	**0.41**	**0.50**	**0.34**	**0.54**	**0.45**	**0.56**	**0.19**	**0.33**	**0.19**	**0.35**	**0.19**	**0.35**

and method level do not match may be that BLUiR puts more emphasis on referred program entity names than ordinary description in bug reports. If a bug report refers to multiple bug-irrelevant methods, BLUiR is misguided to rank methods wrongly. However, once augmented with coverage, BLUiR achieved the highest MAP and MRR values, meaning that coverage improved BLUiR's effectiveness the most significantly.

> **Finding 3:** *Coverage improved IR-based bug localization more significantly at method level than at class level. The average method-level MAP and MRR improvements were 107% and 79%.*

B. RQ2: How does slicing information help with IR-based bug localization?

We experimented with IR_s, and compared their results with those of IR_c. Although JavaSlicer [2] is the best dynamic slicing tool we can use, it has not been maintained for several years. It may not work well for programs requiring features newly introduced in recent JDK versions. Among all the 157 bug fixes, JavaSlicer [2] only ran successfully with 64 examples. For the other examples, JavaSlicer failed for three reasons. First, it threw an out-of-memory exception even though we allocated 8GB memory to JVM. Second, it generated huge traces without termination, violating our 100GB space limit for each subject. Third, the slicing result did not include the actual buggy element due to the tool's limitation when tracing native methods, standard library classes, and multithreaded applications[1]. Therefore, we compared IR_s and IR_c on those 64 bugs for fairness.

Table IV (a) and (b) show the comparison between IR_c and IR_s at both class and method levels. Slicing was more powerful than coverage when improving IR-based bug localization. The reason is slicing removed more irrelevant entities from the IR-based list, and further upgraded ranks of the relevant ones. Similar to the observations in Section V-A, we found that the improvement at method level was more significant than that at class level. For BLUiR, the average MAP and MRR improvements of IR_s over IR_c at method level were 42% and 35%, while the improvements at class level were both 22%. Again, BLUiR achieved the best MAP and MRR when augmented with slicing.

> **Finding 4:** *Slicing was more helpful than coverage in improving IR-based techniques. The average MAP and MRR improvements of IR_s over IR_c were both 15% at class level, with 40% and 30% at method level.*

C. RQ3: How does spectrum information further improve bug localization over IR_c and IR_s?

To evaluate the impact of spectrum on IR_c and IR_s, we enumerated all possible combinations between the four kinds of spectrum information (Section II-B) and IR_c or IR_s. All three basic IR-based techniques were explored for complete comparison. We changed the combination factor α from 0 to 1, with 0.1 increment, to investigate how bug localization effectiveness varies with α.

For IR_{cp}, as shown in Figure 1, we leveraged both coverage and spectrum to improve IR-based techniques. We evaluated MAP and MRR at both class and method levels. *X-axis* represents α. *Y-axis* represents MAP in Figure 1 (a-c) and (g-i), and represents MRR in Figure 1 (d-f) and (j-l). Both MAP and MRR vary within [0, 1]. Intuitively, when $\alpha = 0$, the values are reported for IR_c. When $\alpha = 1$, the reported values are purely from spectrum information. We observed that

[1]The limitations are also listed on JavaSlicer homepage [2].

TABLE IV: IR_c vs. IR_s bug localization

Metric	Project	(a) Class Level						(b) Method Level					
		Baseline		BugLocator		BLUiR		Baseline		BugLocator		BLUiR	
		IR_c	IR_s	IR_c	IR_s	IR_c	IR_s	IR_c	IR_s	IR_c	IR_s	IR_c	IR_s
Top 1	Ant	15	17	14	16	13	17	4	6	3	4	3	6
	Lucene	5	6	6	11	2	6	3	5	5	6	3	5
	Rhino	3	3	2	3	3	4	0	0	0	0	0	0
	Overall	**23**	**26**	**22**	**30**	**18**	**27**	**7**	**11**	**8**	**10**	**6**	**11**
Top 5	Ant	26	26	23	26	23	25	12	17	12	18	10	17
	Lucene	27	28	16	28	22	24	13	14	12	15	11	13
	Rhino	15	18	7	9	10	13	0	0	0	1	0	0
	Overall	**68**	**72**	**46**	**63**	**55**	**62**	**25**	**31**	**24**	**34**	**21**	**30**
Top 10	Ant	27	27	24	28	25	25	16	18	18	24	17	18
	Lucene	32	33	19	34	27	28	16	16	15	18	14	15
	Rhino	20	20	8	11	13	13	0	1	0	1	0	1
	Overall	**79**	**80**	**51**	**73**	**65**	**66**	**32**	**35**	**33**	**43**	**31**	**34**
MAP	Ant	0.71	0.79	0.71	0.75	0.71	0.85	0.37	0.52	0.33	0.34	0.35	0.52
	Lucene	0.50	0.58	0.44	0.62	0.44	0.54	0.30	0.45	0.35	0.45	0.34	0.46
	Rhino	0.47	0.50	0.50	0.50	0.48	0.58	0.02	0.04	0.02	0.11	0.02	0.04
	Mean	**0.56**	**0.62**	**0.55**	**0.62**	**0.54**	**0.66**	**0.23**	**0.34**	**0.23**	**0.30**	**0.24**	**0.34**
MRR	Ant	0.75	0.80	0.75	0.78	0.73	0.86	0.40	0.52	0.38	0.41	0.36	0.52
	Lucene	0.53	0.61	0.49	0.68	0.43	0.57	0.37	0.49	0.44	0.51	0.39	0.50
	Rhino	0.46	0.49	0.50	0.53	0.48	0.58	0.02	0.03	0.02	0.11	0.02	0.034
	Mean	**0.58**	**0.63**	**0.58**	**0.66**	**0.55**	**0.67**	**0.26**	**0.35**	**0.28**	**0.34**	**0.26**	**0.35**

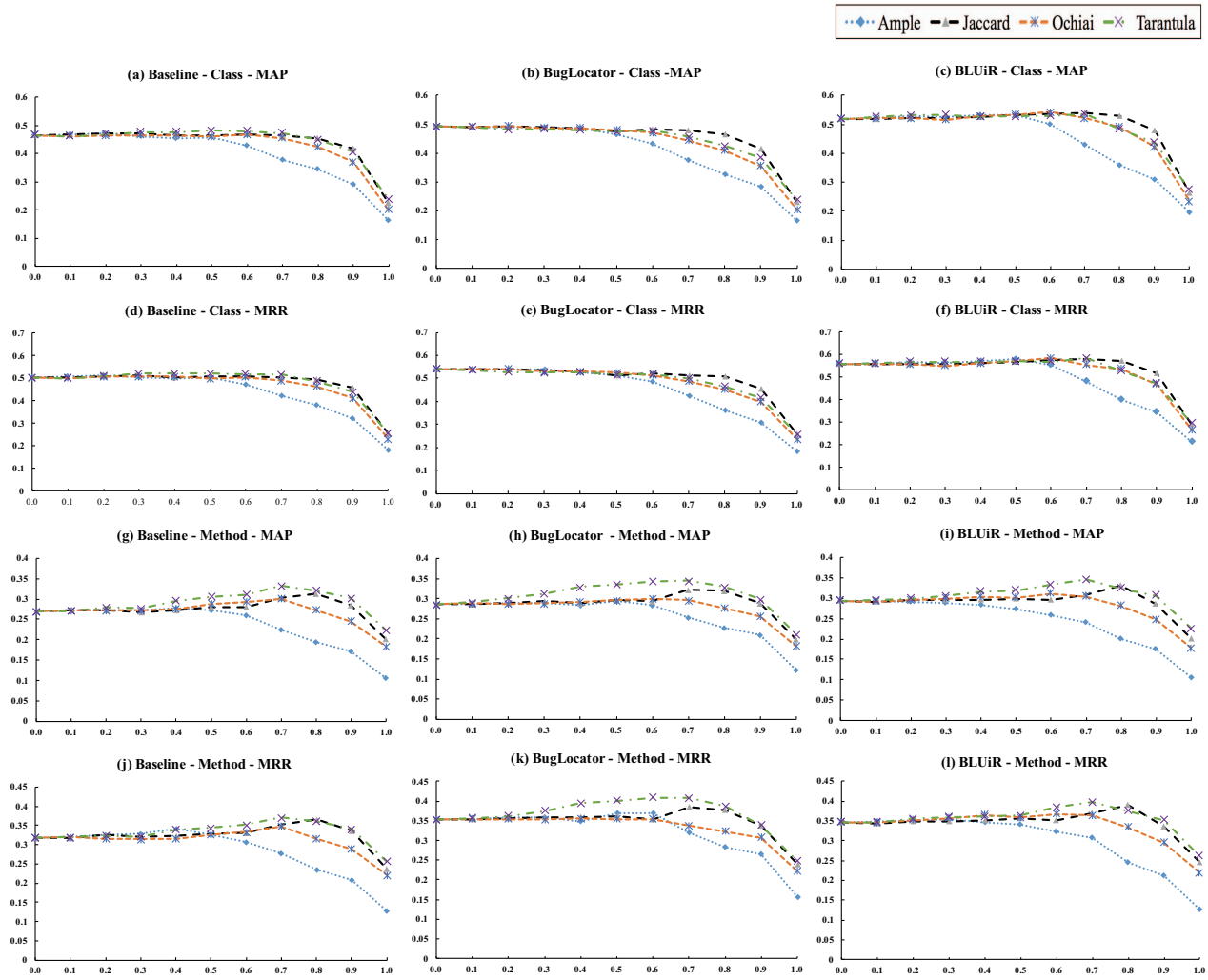

Fig. 1: Effectiveness of IR_{cp} at the class level (a-f) and method level (g-l). The x-axis represents α. The y-axis of (a)-(c) and (g)-(i) represents MAP, while the y-axis in (d)-(f) and (j)-(l) is MRR.

IR_c always worked better than spectrum, because IR_c utilized both static and dynamic information, while spectrum was only

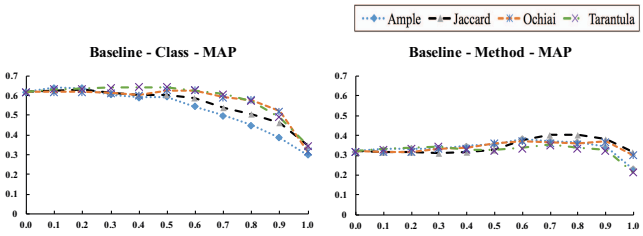

Fig. 2: The MAP of IR$_{sp}$ with the baseline IR technique

dynamic information. The optimal combination between IR$_c$ and spectrum always worked better than either component. Tarantula worked better than other spectrum information.

At class-level, IR$_{cp}$ at most worked *slightly* better than IR$_c$. The reason may be that the class-level spectrum describes execution too coarsely: A class can contain many methods. Whenever a method is executed by a test, the whole class is considered covered. Such class-level spectrum makes suspiciousness scores less helpful.

When using Tarantula with $\alpha = 0.7$ for method-level bug localization, IR$_{cp}$ significantly outperformed IR$_c$. As shown in Figure 1(g-i) and Figure 1(j-l), MAP was increased by 14-22%, while MRR was increased by 16-21%. In particular, RHINO-519692 and its corresponding buggy method only shared one word in common: *transformNewExpr()*, so IR$_c$ ranked the buggy method as 12^{th}, with a low score of 0.28. However, Tarantula considered the method as the most suspicious one, because both of the two tests executing it failed. Therefore, IR$_{cp}$ effectively improved the method's rank as Top 1 by properly combining IR$_c$ with Tarantula spectrum.

We also experimented with IR$_{sp}$, and made similar observations. Due to the space limit, we only show part of the results in Fig. 2. With the baseline IR-based technique, IR$_{sp}$ did not improve over IR$_s$ at class level (on the left), but achieved noticeable improvement at method level (on the right).

> **Finding 5:** *Spectrum information was effective to improve IR-based bug localization at method level instead of at class level. With Tarantula and $\alpha = 0.7$, IR$_{cp}$ and IR$_{sp}$ almost always achieved the best effectiveness.*

D. RQ4: How do our simple combinations compare with the state-of-the-art hybrid technique AML?

We directly took the AML results in prior work [13], and did similar experiments using our hybrid approaches IR$_c$ and IR$_{cp}$. We experimented with the same dataset of 157 examples as AML for fair comparison. Since BLUiR always outperformed other tools when combined with execution information, it was used as the basic IR-based technique in the comparison. IR$_{cp}$ was configured to use Tarantula spectrum with $\alpha = 0.7$, because the setting was demonstrated the best in Section V-C.

As shown in Table V, we found that AML ranked more Top-1 entities correctly than IR$_c$ (31 vs. 29). Other than that, IR$_c$ worked better in terms of Top-5, Top-10, and MAP metrics. The ***overall*** values of IR$_{cp}$ were better than AML for all metrics. Although we did not compare IR$_s$ and IR$_{sp}$ directly with AML due to the JavaSlicer limitation, it is reasonable to expect both of them to perform better than AML. The reason is compared with coverage, slicing always identifies failure-relevant entities more precisely, and refines IR-based ranked lists more effectively. Our experiments with a subset of the data did demonstrate that IR$_s$ worked better than IR$_c$, and IR$_{sp}$ worked better than IR$_{cp}$.

The fact that our simple combination approaches worked better than AML reveals two things. First, all kinds of dynamic information can effectively improve IR-based bug localization. Second, a complex combination approach is not always necessary to better localize bugs.

TABLE V: Bug localization comparison with AML at method level

Metric	Project	IR	IR$_c$	IR$_{cp}$	AML
Top 1	AspectJ	2	3	8	7
	Ant	6	13	14	9
	Lucene	5	7	6	11
	Rhino	5	6	8	4
	Overall	**18**	**29**	**36**	**31**
Top 5	AspectJ	4	7	14	13
	Ant	16	36	39	22
	Lucene	16	32	23	22
	Rhino	7	15	22	14
	Overall	**43**	**90**	**98**	**71**
Top 10	AspectJ	6	12	19	13
	Ant	25	54	57	31
	Lucene	24	42	30	29
	Rhino	10	22	27	19
	Overall	**65**	**130**	**133**	**92**
MAP	AspectJ	0.06	0.10	0.22	0.19
	Ant	0.15	0.36	0.37	0.23
	Lucene	0.15	0.38	0.30	0.28
	Rhino	0.19	0.34	0.50	0.24
	Overall	**0.14**	**0.30**	**0.35**	**0.24**

> **Finding 6:** *IR$_c$ outperformed AML in all metrics but one, while IR$_{cp}$ outperformed AML. Our simple approaches worked better than AML's more complicated approach, showing that various execution information can effectively help with IR-based bug localization.*

VI. Discussion

Our empirical study demonstrates that various kinds of execution information can *effectively* improve IR-based bug localization, as long as the information usage is *proper, but not necessarily complex*. In the study, there are still bug reports whose bugs are not located by any investigated approach. We further examined these reports and their bugs.

Among the 157 bug reports, 18 reports contain no clue about where the bug is, 77 reports mention bug-relevant code elements, such as fields or methods inside the buggy classes, and 62 reports have the exact buggy class names explicitly mentioned. For those reports without any clue of bug locations, a bug reporter usually describes the bug-triggering input(s) or bug symptoms, and the execution information does not help reveal bugs, either. Although such bugs do not count much in our dataset, they may be prevalent in reality, and require more advanced novel solutions.

For some reports with either buggy classes explicitly mentioned, or bug-relevant information (i.e. fields or methods in buggy classes) included, the investigated approaches failed for two reasons. First, some mentioned buggy entity names are so widely used that they are not distinctive, such as "*set*" and "*method*". Second, when test cases or call stacks include many entities to describe the problems, the noisy location information confuses IR-based techniques.

In future, we will integrate static analysis-based fault prediction [7] techniques to better localize bugs. For example, when a buggy method has a popular name like "set", and is covered by both passed and failed tests, neither the bug report nor execution information is helpful. We can use fault prediction to calculate various metrics (e.g., *fan-in/fan-out*[2]) to measure how likely each method is buggy. By ranking methods based on their fault prediction scores, we will obtain a ranked list, which can be further combined with the list produced by a hybrid approach of IR-based bug localization and execution information.

VII. Threats to Validity

We reused an existing dataset of 157 real bugs from 4 open source projects to evaluate different bug localization techniques. The evaluation results may not generalize to other bugs or other open source projects. The collected execution information also strongly depends on the quality and availability of test cases.

We collected slicing information with JavaSlicer [2], because the tool is the only publicly available dynamic slicing tool based on our knowledge. The limitation of the tool may affect the generalizability of our observations. Besides, we used four most popular formulae to calculate spectrum, and used three IR-based bug localization techniques. The limited number of investigated formulae and IR-based techniques may also affect the generalizability.

VIII. Related Work

In this section, we will discuss related work in spectrum-based fault localization, IR-based bug localization, and empirical studies on bug localization techniques. Different from all prior work, we systematically experimented with three IR-based techniques, three kinds of execution information, and investigated four ways to combine them for both class-level and method-level bug localization. Our study generally reveals that different kinds of execution information can considerably improve IR-based bug localization, no matter what particular technique is integrated or whether the combination algorithm is simple or complicated.

Spectrum-Based Fault Localization (SBFL) identifies bug locations using the execution information of buggy code [9], [18], [33], [4], [5], [30]. Given a buggy program and test cases, SBFL instruments code to collect the execution information of passed tests and failed tests, and counts how many passed and failed tests execute each program element (i.e., class, method, or statement). SBFL then calculates a suspiciousness score for each element to find the most suspicious element(s). Different formulae have been defined to compute suspiciousness scores, such as Tarantula [9], Ochiai [4], and Ample [5]. Xuan et al. used machine learning to train a suspiciousness model by combining multiple existing formulae [30]. Our experiments with four most popular SBFL techniques showed that merging SBFL with IR-based bug localization can improve SBFL, which aligns with prior finding [13].

IR-Based Bug Localization localizes bugs based on bug reports [36], [21], [32], [12]. A basic technique treats a bug report as a query and source code as documents, and retrieves the documents most relevant to the query as bug locations. More advanced techniques leverage additional information to better localize bugs. For instance, BugLocator integrates knowledge of previously fixed similar bugs [36]. BLUiR integrates knowledge of programs or bug report structures to assign more weight to class and method names, and bug report titles [21]. Learning-to-rank integrates domain knowledge of bug history and API specification to train a model for bug location prediction [32]. HyLoc leverages Deep Neural Network(DNN) to learn a model, that correlates bug reports with code tokens (i.e., identifiers, APIs), and textual tokens (i.e., comments) in code [12]. However, none of these techniques integrate program execution information. Our investigation demonstrates that hybrid approaches can generally work better than pure IR-based approaches.

Empirical Studies on Bug Localization Techniques have been done by researchers [14], [34], [27], [11]. Lucia et al. [14] and Yoo et al. [34] compared various formulae used in SBFL, and observed that there was no best formula that consistently outperformed others. Kochhar et al. manually examined bug reports whose bugs were either fully, partially, or not localized by IR-based techniques [11]. They found that the quality of bug reports can substantially impact the effectiveness of IR-based techniques. If bug reports explicitly contain buggy file names, IR-based techniques are more likely to locate the bugs. Wang et al. conducted a user study with developers to examine whether IR-based techniques actually help developers localize bugs [27]. The study showed that IR-based techniques were not always useful. Our empirical study complements the observations by prior studies. We investigated different ways to combine various execution information with IR-based bug localization techniques. We explored whether execution information generally helps with IR-based techniques, and whether a complicated combination algorithm is required to improve IR-based techniques with dynamic information.

IX. Conclusions

It is challenging to locate bugs given bug reports. In this empirical study, we investigated how various dynamic execution information (e.g., coverage, slicing, and spectrum information) can help with IR-based bug localization. We found that through refining the ranked list of suspicious locations produced by IR-based techniques, coverage and slicing information can effectively help improve bug localization.

[2]Fan-in denotes the number of methods invoking the method, while fan-out denotes the number of methods invoked by the method.

Spectrum information can further improve method-level bug localization by merging its suspicious location list with the coverage-refined or slicing-refined IR-based list.

Our investigation with the three types of execution information demonstrates that dynamic information can effectively improve IR-based bug localization, even though the information is integrated in simple ways. By comparing our combination approaches with a state-of-the-art hybrid technique of IR-based bug localization and spectrum, we observed that our simple approaches almost always worked better. It means that a combination approach does not have to be complicated for effective bug localization. When examining bugs that none of the investigated techniques can handle, we found it promising to conduct and combine static analysis-based fault prediction to further better bug localization.

ACKNOWLEDGMENT

We thank anonymous reviewers for their thorough comments on our earlier version of the paper. This work was partially supported by NSF Grant No. CCF-1565827 and CCF-1566589, and Google Faculty Research Award.

REFERENCES

[1] ASM. http://asm.ow2.org.

[2] Javaslicer. https://www.st.cs.uni-saarland.de/javaslicer/.

[3] R. Abreu, P. Zoeteweij, and A. J. C. v. Gemund. An evaluation of similarity coefficients for software fault localization. In *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing*, PRDC '06, pages 39–46, Washington, DC, USA, 2006. IEEE Computer Society.

[4] R. Abreu, P. Zoeteweij, and A. van Gemund. On the accuracy of spectrum-based fault localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007*, pages 89–98, Sept 2007.

[5] V. Dallmeier, C. Lindig, and A. Zeller. Lightweight bug localization with ample. In *Proceedings of the Sixth International Symposium on Automated Analysis-driven Debugging*, 2005.

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

[7] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *Software Engineering, IEEE Transactions on*, 38(6):1276–1304, 2012.

[8] D. A. Hull. Stemming algorithms: A case study for detailed evaluation. *J. Am. Soc. Inf. Sci.*, 47(1):70–84, Jan. 1996.

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

[10] J. A. Jones, M. J. Harrold, and J. Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th International Conference on Software Engineering*, 2002.

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

[12] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen. Combining deep learning with information retrieval to localize buggy files for bug reports (n). In M. B. Cohen, L. Grunske, and M. Whalen, editors, *ASE*, pages 476–481. IEEE, 2015.

[13] T.-D. B. Le, R. J. Oentaryo, and D. Lo. Information retrieval and spectrum based bug localization: Better together. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 579–590, New York, NY, USA, 2015. ACM.

[14] Lucia, D. Lo, L. Jiang, and A. Budi. Comprehensive evaluation of association measures for fault localization. In *Software Maintenance (ICSM), 2010 IEEE International Conference on*, pages 1–10, Sept 2010.

[15] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn. Bug localization using latent dirichlet allocation. *Information and Software Technology*, 52(9):972 – 990, 2010.

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

[17] A. Marcus and J. I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering*, 2003.

[18] L. Naish, H. J. Lee, and K. Ramamohanarao. A model for spectra-based software diagnosis. *ACM Trans. Softw. Eng. Methodol.*, 20(3):11:1–11:32, Aug. 2011.

[19] A. T. Nguyen, T. T. Nguyen, J. M. Al-Kofahi, H. V. Nguyen, and T. N. Nguyen. A topic-based approach for narrowing the search space of buggy files from a bug report. In P. Alexander, C. S. Pasareanu, and J. G. Hosking, editors, *ASE*, pages 263–272. IEEE Computer Society, 2011.

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

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

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

[23] B. Sisman and A. C. Kak. Incorporating version histories in information retrieval based bug localization. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, MSR '12, pages 50–59, Piscataway, NJ, USA, 2012. IEEE Press.

[24] T. Strohman, D. Metzler, H. Turtle, and W. B. Croft. Indri: A language model-based search engine for complex queries. *Proceedings of the International Conference on Intelligence Analysis*, 2004.

[25] F. Tip. A survey of program slicing techniques. Technical report, Amsterdam, The Netherlands, The Netherlands, 1994.

[26] J. M. Voas. Pie: a dynamic failure-based technique. *IEEE Transactions on Software Engineering*, 18(8):717–727, Aug 1992.

[27] Q. Wang, C. Parnin, and A. Orso. Evaluating the usefulness of ir-based fault localization techniques. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, pages 1–11, New York, NY, USA, 2015. ACM.

[28] S. Wang and D. Lo. Version history, similar report, and structure: Putting them together for improved bug localization. In *Proceedings of the 22Nd International Conference on Program Comprehension*, ICPC 2014, pages 53–63, New York, NY, USA, 2014. ACM.

[29] M. Weiser. Program slicing. In *Proceedings of the 5th International Conference on Software Engineering*, 1981.

[30] J. Xuan and M. Monperrus. Learning to combine multiple ranking metrics for fault localization. In *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*, pages 191–200, Sept 2014.

[31] J. Xuan and M. Monperrus. Test case purification for improving fault localization. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 52–63, 2014.

[32] X. Ye, R. Bunescu, and C. Liu. Learning to rank relevant files for bug reports using domain knowledge. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 689–699, New York, NY, USA, 2014. ACM.

[33] S. Yoo. Evolving human competitive spectra-based fault localisation techniques. In *Proceedings of the 4th International Conference on Search Based Software Engineering*, SSBSE'12, pages 244–258, Berlin, Heidelberg, 2012. Springer-Verlag.

[34] S. Yoo, X. Xie, F.-C. Kuo, T. Y. Chen, and M. Harman. No pot of gold at the end of program spectrum rainbow: Greatest risk evaluation formula does not exist. Technical report, University College London and Swinburn University, 2014.

[35] Y. Zhang and A. Mesbah. Assertions are strongly correlated with test suite effectiveness. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE*, pages 214–224, 2015.

[36] 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 *Software Engineering (ICSE), 2012 34th International Conference on*, pages 14–24, June 2012.