

# Pathidea: Improving Information Retrieval-Based Bug Localization by Re-Constructing Execution Paths Using Logs

An Ran Chen<sup>✉</sup>, Student Member, IEEE, Tse-Hsun Chen<sup>✉</sup>, Member, IEEE,  
and Shaowei Wang<sup>✉</sup>, Member, IEEE

**Abstract**—To assist developers with debugging and analyzing bug reports, researchers have proposed information retrieval-based bug localization (IRBL) approaches. IRBL approaches leverage the textual information in bug reports as queries to generate a ranked list of potential buggy files that may need further investigation. Although IRBL approaches have shown promising results, most prior research only leverages the textual information that is “visible” in bug reports, such as bug description or title. However, in addition to the textual description of the bug, developers also often attach logs in bug reports. Logs provide important information that can be used to re-construct the system execution paths when an issue happens and assist developers with debugging. In this paper, we propose an IRBL approach, Pathidea, which leverages logs in bug reports to re-construct execution paths and helps improve the results of bug localization. Pathidea uses static analysis to create a file-level call graph, and re-constructs the call paths from the reported logs. We evaluate Pathidea on eight open source systems, with a total of 1,273 bug reports that contain logs. We find that Pathidea achieves a high recall (up to 51.9 percent for Top@5). On average, Pathidea achieves an improvement that varies from 8 to 21 and 5 to 21 percent over BRTracer in terms of Mean Average Precision (MAP) and Mean Reciprocal Rank (MRR) across studied systems, respectively. Moreover, we find that the re-constructed execution paths can also complement other IRBL approaches by providing a 10 and 8 percent improvement in terms of MAP and MRR, respectively. Finally, we conduct a parameter sensitivity analysis and provide recommendations on setting the parameter values when applying Pathidea.

**Index Terms**—Bug localization, log, bug report, information retrieval

## 1 INTRODUCTION

SOFTWARE debugging is one of the most time-consuming tasks in software maintenance. On average, developers spend 33 percent of their time on debugging and bug fixing, rather than on implementing new features [1]. When an issue occurs, developers would create a bug report that documents the necessary information for others to reproduce, diagnose, and fix the bug. However, due to limited time and resources, many bugs remain unfixed for a long period of time. A prior study [2] finds that it often takes several months for developers to address a bug report, which further hinders the user-perceived quality of the system.

To help developers speed up the debugging process, researchers have proposed various information retrieval-based bug localization (IRBL) approaches [3], [4], [5], [6], [7], [8], [9], [10], [11]. IRBL approaches leverage information retrieval to help developers locate potentially buggy files.

Given a bug report, IRBL approaches use its textual information as queries to generate a ranked list of source code files, based on their textual similarity, that are potentially buggy. To improve the performance of IRBL, researchers have proposed approaches that utilize various information in software repositories, such as similar bug reports from the past [3], software development history [11], [12], and structured information in bug reports [6].

In addition to the textual description of a bug, developers often provide the system execution information when a bug happens. Developers may attach log snippets (e.g., 2015-07-01 19:24:12,806 INFO org.apache.ZooKeeper.ClientCnxn: Client session timed out) or stack traces (e.g., java.lang.NullPointerException) that show a snapshot of the system execution. Prior studies [13], [14], [15] show that logs (i.e., log snippets or stack traces) can be mapped to source code to re-construct execution paths and assist developers with debugging. Even though a number of IRBL approaches [3], [16] try to leverage stack traces in bug reports, they only analyze the file names that appear directly in stack traces to boost the bug localization performance. Yet, the embedded system execution information, which can be re-constructed by linking the logs to their corresponding location in the source code, may further help improve the performance of IRBL approaches.

In this paper, we propose Pathidea, an IRBL approach that leverages the execution paths which are re-constructed from logs in bug reports. Pathidea uses static analysis to

- An Ran Chen and Tse-Hsun Chen are with Software PErformance, Analysis, and Reliability (SPEAR) Lab, Concordia University, Montreal, QC H3G 1M8, Canada. E-mail: {anr\_chen, peterc}@encs.concordia.ca.
- Shaowei Wang is with the Department of Computer Science, University of Manitoba, Winnipeg, MB R3T 2N2, Canada. E-mail: shaowei@cs.umanitoba.ca.

Manuscript received 22 July 2020; revised 16 Mar. 2021; accepted 3 Apr. 2021.  
Date of publication 6 Apr. 2021; date of current version 15 Aug. 2022.  
(Corresponding author: An Ran Chen.)  
Recommended for acceptance by W. Visser.  
Digital Object Identifier no. 10.1109/TSE.2021.3071473

create a file-level call graph, and uncover the path in the call graphs that may be related to the bugs. Different from prior studies which only utilize stack traces from bug reports, Pathidea analyzes both log snippets and stack traces. Similar to stack traces, log snippets also contain important information that can be used to re-construct the execution paths when an issue happens, and thus, help with bug localization.

We evaluate Pathidea on eight open source Java systems, with a total of 1,273 bug reports that contain logs (i.e., log snippets, stack traces, or both). We first compare Pathidea with two baseline IRBL approaches (i.e., VSM and BRTtracer [16]) using five commonly used metrics (i.e., precision, recall, and F1-measure for Top@N, MAP, and MRR). Our results show that, while both Pathidea and BRTtracer outperform the vanilla VSM in identifying buggy files, Pathidea achieves an average improvement that varies from 8 to 21 and 5 to 21 percent over BRTtracer in terms of MAP and MRR, respectively. Then, we study if the re-constructed execution paths can provide additional improvement to existing IRBL approaches (i.e., BRTtracer). Our results show that when integrating the path analysis (i.e., linking the logs to their corresponding location in the source code and re-constructing the execution paths accordingly) with BRTtracer, the execution paths information can provide a 10 and 8 percent improvement in terms of MAP and MRR, respectively. To the best of our knowledge, Pathidea is the first IRBL approach to utilize the re-constructed execution paths based on logs provided in bug reports. Our case study shows that Pathidea can identify buggy files with high recall values (i.e., up to 51.9 and 57.7 percent for Top@5 and Top@10, respectively), and the path analysis may also complement other IRBL approaches. In summary, the contributions of this paper are as follows:

- We proposed Pathidea, a new IRBL approach that uses static analysis to re-construct execution paths from logs in bug reports to help locate the potential buggy files. To the best of our knowledge, Pathidea is the first approach that incorporates the re-constructed execution paths into the IRBL approach.
- We conducted a case study to evaluate Pathidea on eight open source systems. The results demonstrate that Pathidea can identify buggy files with high precision and recall values, and outperforms existing state-of-the-art IRBL approaches.
- Our results show that the re-constructed execution paths can complement existing IRBL approaches by improving bug localization performance.
- We conducted a parameter sensitivity analysis and provide recommendations on setting the parameter values when applying Pathidea.

In summary, our approach sheds light on further improving IRBL approaches by combining information in bug reports with the source code. Future studies may consider leveraging such execution paths information when designing IRBL approaches. The data of our experiment is publicly available online.<sup>1</sup>

*Paper Organization.* In Section 2, we present some background information on bug reports and information

retrieval, motivating examples, and related work. In Section 3, we describe our approach in detail. In Section 4, we explain our data collection process and the evaluation metrics. In Section 5, we present our results and the findings for each research question. In Section 6, we further elaborate on the results. In Section 7, we discuss the threats to validity. Finally, Section 8 concludes the paper.

## 2 BACKGROUND AND RELATED WORK

Bug reports contain various fields that help reporters better describe the bug. There are three main fields that provide important information for developers to debug the problem: summary, detailed description, and comments. The summary field provides a short description of the bug, while the detailed description expands on the encountered problem. The reporters typically provide the steps to reproduce the bug, observed behaviors, stack traces, or log snippets in the detailed description to help reproduce and fix the bug. When further clarifications are needed, developers may ask the reporters to provide additional information in the comments field [17].

The textual information in bug reports often provide hints on where the bugs may be located. To help developers reduce the needed time for locating the bugs, researchers have proposed a series of information-retrieval based bug localization (IRBL) approaches [3], [4], [5], [6], [7], [8], [9], [10], [11]. IRBL approaches compute the textual similarity between the bug report and source code files using approaches from information retrieval, such as vector space model (VSM). VSM is an algebraic model that represents documents (i.e., bug reports and source code files) as vectors of index terms. Each document is represented as a term-frequency vector in an  $n$ -dimensional space, where  $n$  denotes the number of unique terms in the corpus (i.e., collection of documents). When applying VSM in IRBL, a bug report represents the search query, while the entire source code files are used as the corpus, with the goal of finding the document (i.e., source code file) that has the highest textual similarity (e.g., largest cosine similarity) with the given bug report.

Although IRBL approaches have shown promising results, most of the prior studies only treat the information in bug reports as pure text. However, in addition to the textual description in bug reports, developers also heavily rely on the logs that the reporters provide to understand and debug issues [18]. Logs, either log snippets or stack traces, show the partial system execution when a problem occurs. Prior studies [13], [14] show that logs can be mapped to source code and assist developers with understanding the system execution during debugging and maintenance. Such valuable information may further help improve the performance of IRBL approaches. Fig. 1 depicts the stack trace extracted from the Description section of a bug report from YARN. Based on the textual information in the stack trace, IRBL approaches may identify files such as DockerClient, DockerCommandExecutor, LinuxContainerExecutor, and DockerContainerDeletionTask (i.e., the name of the files shown in the stack trace) as potentially buggy files. However, the information is limited as we may overlook what happens between each stack frame. To resolve this bug, the developers provided a fix to the PrivilegedOperation file (shown in Fig. 2), which is called

1. [https://github.com/SPEAR-SE/Pathidea\\_Data](https://github.com/SPEAR-SE/Pathidea_Data)

<b>Bug ID</b>	YARN-8209	
<b>Summary</b>	NPE in DeletionService	
<b>Stack Trace</b>	<pre>2018-04-25 23:38:41,039 WARN concurrent.ExecutorHelper (ExecutorHelper.java:logThrowableFromAfterExecute(63)) :   ↪ java.lang.NullPointerException   at DockerClient.writeCommandToTempFile (DockerClient.java:109)   at DockerCommandExecutor.executeDockerCommand (DockerCommandExecutor.java:3)   at DockerCommandExecutor.executeStatusCommand (DockerCommandExecutor.java:192)   at DockerCommandExecutor.getContainerStatus (DockerCommandExecutor.java:128)   at LinuxContainerExecutor.removeDockerContainer (LinuxContainerExecutor.java:935)   at DockerContainerDeletionTask.run (DockerContainerDeletionTask.java:61)   at java.lang.Thread.run (Thread.java:748)</pre>	

Fig. 1. Stack traces extracted from the bug report YARN-8209.

during the execution shown in the second last stack frame (`at DockerCommandExecutor.executeDockerCommand(DockerCommandExecutor.java : 3)`), but not included in the stack trace (i.e., the call to `PrivilegedOperation` is already popped from the stack). Therefore, such *hidden execution* is invisible to IRBL approaches. Thus, in this paper, we aim to utilize the execution paths information that can be re-constructed from logs to provide more information and improve bug localization performance.

A number of studies have been conducted on multiple faults localization [19], [20], [21]. Wong *et al.* [19] performed a detailed survey on the software multiple faults localization techniques. Zakari *et al.* [20] conducted a systematic literature review on multiple faults localization. Pearson *et al.* [21] evaluated the effectiveness of various multiple faults localization techniques on artificial faults (e.g., mutants) versus real faults. In this paper, we primarily focus on single fault localization. However, if there were logs generated from a program with multiple faults, our approach can potentially be applied in existing multiple faults localization techniques to improve their effectiveness.

Below, we further discuss the related work of this paper.

*Information-Based Bug Localization Approaches.* Many prior studies apply information retrieval (IR) approaches to statically locate bugs in the code using natural language text in bug reports [3], [4], [5], [6], [7], [8], [10], [11]. Zhou *et al.* [3], proposed an approach, called BugLocator, that locates buggy files by leveraging historical similar bug reports. They compute a suspiciousness score between a bug report and the

source code files based on their textual similarity. Loyola *et al.* [8] and Sisman and Kak [11] found that by using past version development history can help improve bug localization accuracy. Wang and Lo [6] and Saha *et al.* [4] found that different parts of bug reports (e.g., title and description) should be assigned different weights in IR models. Liu *et al.* [7] combine statistical debugging and dynamic model slicing on top of Simulink models to improve bug localization accuracy. Bhagwan *et al.* [10] applied differential code analysis to pin-point the buggy commits in the development history. Lam *et al.* [5] combine VSM with the deep neural network to improve the performance of IR-based bug localization approaches. They address the lexical mismatch problem by connecting the terms in bug reports to their related but different code tokens in source code files.

Different from prior studies, in this paper, we propose an approach that uses the execution paths information re-constructed from logs to assist IRBL approaches. We find that our path analysis approach is complementary to existing IRBL approaches and improve bug localization performance.

*IRBL Approaches Using Execution Information in Bug Reports.* Some prior studies leverage the system execution information (e.g., stack traces or test case execution) in bug reports to help locate buggy files. Dao *et al.* [9] leveraged the coverage, slicing, and spectrum information in failed test cases to improve IR-based bug localization. Wong *et al.* [16] propose BRTtracer, a bug-report-oriented fault localization tool. The tool is built on top of BugLocator [3] and uses the file names that appear in stack traces to further rank the suspicious files. They studied 3,459 bug reports across three systems in which only 17 percent of the bug reports include stack traces. In addition to stack traces, Youm *et al.* [22] propose a new bug localization approach, BLIA (Bug Localization using Integrated Analysis) that integrates analyzed data by utilizing structured information in bug reports and source code files, code change history, similarity analysis of existing bug reports, and stack traces.

Prior studies only leverage the “visible” information in bug reports (e.g., stack traces). However, as shown in Figs. 1 and 2, sometimes the hidden execution paths that can be re-constructed in bug reports can provide additional information for debugging and bug localization. The most similar work is done by Chen *et al.* [15], [23], where they study if the logs in bug reports can be used to locate buggy files. They find that logs in bug reports provide a good indication on where the buggy files, although the provided logs may be outdated or

```
1 public class DockerCommandExecutor {
2
3     public static String executeDockerCommand
4         (DockerCommand dockerCommand, ...) throws
5             ContainerExecutionException {
6
7         PrivilegedOperation dockerOp =
8             dockerCommand.preparePrivilegedOperation
9             (dockerCommand, ...);
10
11         if (disableFailureLogging) {
12             dockerOp.disableFailureLogging();
13         }
14     }
15 }
```

Fig. 2. Simplified source code from `DockerCommandExecutor.executeDockerCommand`. The fix of YARN-8209 was applied in `PrivilegedOperation`.

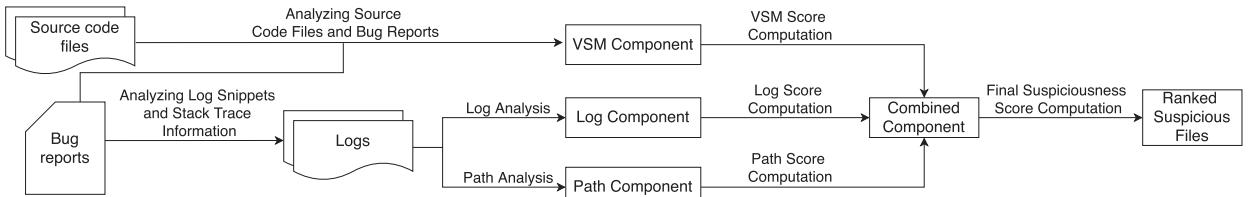


Fig. 3. An overview of Pathidea.

can no longer be found in the source code. In this paper, we leverage the stack traces and log snippets in bug reports to reconstruct the system execution paths to complement IRBL approaches. We propose an IRBL approach called Pathidea and we find that it achieves better performance compared to state-of-the-arts such as BRTracer. Moreover, we find that the hidden execution information re-constructed from our path analysis can also help improve the performance of existing IRBL approaches (see details in Sections 5.1 and 5.2).

### 3 METHODOLOGY

In this section, we first present an overview of Pathidea. Then, we describe each step of our approach in detail.

*An Overview of Pathidea.* Fig. 3 shows an overview of our approach, which contains four major steps: (1) In the source code analysis step, we build a Vector Space Model (VSM) and compute an initial similarity score between the bug report and the source code files. We denote this initial score as *VSM score*. (2) In the log analysis step, we highlight the related files that appear directly in the logs using regular expression. Depending on the type of log, we apply different strategies to derive a boost score (i.e., to adjust the weight of the files), denoted as *log score*. (3) In the path analysis step, we re-construct the file-level execution paths from the logs to find the files which were called during the execution time. We assign a new boost score, which we denote as the *path score*, to these files. (4) Finally, we add the log and path scores into our initial similarity score to calculate the final suspiciousness score of a file. We rank the files based on the suspiciousness score and derive a list of ranked files for investigation. Below, we discuss the aforementioned steps in detail.

*Analyzing Source Code Files and Bug Reports.* To analyze the source code files and bug reports, we follow common source code pre-processing steps [24]. We first tokenize the source code file into a series of lexical tokens and remove programming language specific keywords [25] (e.g., *for* and *while* for Java). Next, we split concatenated words based on camel case (e.g., *getAverage*) and underscore (e.g., *get\_average*) and remove stopwords (e.g., *the* and *and*). We use the list of stopwords from the Natural Language Toolkit (NLTK) library in Python [26]. Finally, we perform Porter stemming to remove morphological affixes from words and derive their common base form (e.g., *running* becomes *run*). As mentioned in Section 2, the output of this process is a collection of corpus, where each document represents a source code file. Given a bug report, we extract the lexical tokens from the *summary* and *description* fields. To represent each bug report as a search query, we follow the same pre-processing steps described above.

Since larger files contain more tokens, by nature, large files are more likely to be favored in bug localization [16]. Thus, to treat all files equally regardless of its size, we follow a prior study [16] by using a segmentation approach when creating the corpus. The segmentation approach divides each file into multiple segments of code snippets of the same size. Namely, each document in the corpus represents a segment of code snippets from a source code file. Then, given a bug report, the corresponding file of the segment that has the highest suspiciousness score is marked as the most suspicious file for investigation. Similar to the study by Wong *et al.* [16], we set the segment size to 800 tokens.

More specifically, Formula 1 below calculates the VSM score between a file  $f$  and a bug report  $br$ , where  $suspiciousness_{\max}(seg, br)$  is the maximum cosine similarity score between all the segments  $seg$  in  $f$  and  $br$ .

$$\text{VSM Score}(f, br) = \text{suspiciousness}_{\max}(seg, br). \quad (1)$$

*Analyzing Log Snippets and Stack Trace Information.* Logs provide an important source of information to developers. Prior studies [1], [13], [27], [28] have shown that developers often leverage logs to understand how the system was executed for debugging and testing purposes. Thus, our approach aims to utilize both types of logs (i.e., log snippets and stack traces) to further assist bug localization. We compute additional suspiciousness scores for the files that generate the logs. We denote the additional suspiciousness score computed from the logs as the log score.

To analyze the logs, we first capture them from a bug report using regular expressions. In particular, for stack traces, we check for the *at* keyword followed by a file name that ends with *.java*. For log snippets, we look for a timestamp followed by a verbosity level and a fully qualified class name. For instance, given the following log line 2015-07-01 19:24:12, 806 INFO org.apache.ZooKeeper.ClientCnxn: Client session timed out, our regular expression captures 2015-07-01 19:24:12, 806 as the timestamp, INFO as the verbosity level, and org.apache.ZooKeeper.ClientCnxn as the fully qualified class name. We use the fully qualified class name to derive its corresponding file name. Next, we verify in the source code repository whether the file exists. This helps us remove the files that are part of the external libraries. Finally, we calculate the log score differently for files extracted from stack traces and from log snippets. For stack trace, we use the rank of the file in the call stack to assess its suspiciousness score by following a prior study [16]. If a file appears on the top of the stack trace, it is ranked the first and receives a higher suspiciousness score. Given a rank position  $i$ , if  $i$  is within the top 10 ranks, the suspiciousness score is inversely proportional to the rank (e.g., the second ranked file

receives a suspiciousness score of 0.5). For any rank position  $i$  beyond top 10, the file receives a constant suspiciousness score of 0.1. Formula (2) below calculates the log score for files in a stack trace.

$$\text{LogScore}(f) = \begin{cases} \frac{1}{rank} & \text{if } rank \leq 10 \\ 0.1 & \text{if } rank > 10 \\ 0 & \text{if file not found} \end{cases}. \quad (2)$$

For log snippets, we assign a constant value of 0.1 to every mapped file. We denote this constant value as  $\alpha$ . We use  $\alpha$  as a parameter to attribute a suspiciousness score to each file mapped from log snippets. In RQ3, we further investigate the sensitivity of the value for  $\alpha$ .

When multiple stack traces are attached in a bug report, we regard them as equally valuable. Therefore, to further refine our approach, we reset the rank back to 1 when a new stack trace begins. In log snippets, when the same file appears multiple times, it is only computed once in the log score. Fig. 4 shows an example of the log score computation. The logs start with a log snippet containing two log lines: `task_r_1 done copying task_m_0` and `task_r_1 Copying task_m_1`. As both lines are generated by the same file, that is `ReduceTask.java`, the log score is only computed once with a constant value of 0.1. Two stack traces follow the log snippet. The first stack trace throws a `java.lang.OutOfMemoryError`, where the file `SequenceFile.java` appears in the first stack frame. Therefore, it is ranked as the first place, and its log score is 1.00. Similarly, the file in the second stack frame (i.e., `SequenceFile.java`) receives a log score of 0.50. The second stack trace throws a `java.lang.NullPointerException`, in which `InMemoryFileSystem.java`, `FileSystem.java` and `ReduceTask.java` receive their respective log score based on their order in the stack frames (i.e., 1.00, 0.50 and 0.33, respectively).

*Analyzing and Re-Constructing Execution Paths.* As discussed in Sections 1 and 2, most prior studies only consider the textual information that is available in the bug report. Since logs can be further mapped to the source code, there may be valuable information in the source code that can help developers to better understand the system execution. To this end, we analyze the logs and re-construct the potential execution paths. We describe the steps as follow.

We first extract the related methods and files that appear in the logs. We verify that such methods and files exist in the source code repository. Then, for each related method call, we derive the method-level Abstract Syntax Tree (AST) using Javaparser [29]. Javaparser is a static analysis tool that supports many Java versions and is actively maintained. The AST tree allows us to traverse the AST nodes and find the method calls inside each method declaration. We statically construct the execution path from the AST tree of the method by linking each method call into its method declaration. If a related method call appears in the execution path, we mark it as *visited*. The execution path continues to expand until the last method call in the log is *visited*. Once the execution path is reconstructed, we analyze the execution order of the related

method calls and uncover the potential execution paths. Algorithm 1 shows the pseudo code of our implementation. The algorithm takes extracted logs from bug reports as input, and outputs the potential execution paths. First, we initiate a global variable (line 2) `executionPaths` to store the re-constructed execution path. When we iterate through the logs, we assign the current log at position  $i$  and the next log at position  $i+1$  (line 4-5). Then, the execution paths are derived from the logs (line 6). The `findPathsBetween` function essentially implements the Breadth-First-Search (BFS) algorithm to traverse the call graph. In this process, we record every possible path that connects the current log to the next log. If two consecutive logs are identical (i.e., have the same log template), we remove one from the logs (e.g., the logs may be generated in a loop). Once the execution path is re-constructed, we store it in the local variable `paths` (line 6), which is then added to the global variable `executionPaths` (line 7). Lastly, we return the global variable `executionPaths` (line 9).

Note that a path is constructed for each sequential set of logs (e.g., logs belong to the same thread). Thus, after we have obtained the re-constructed execution paths, there may be some duplicated paths due to the looping of some logs generated at runtime by different threads. Therefore, we compare the sequence of method calls inside each generated path and remove the duplicated paths.

In our experiment, we use a virtual machine, with a four-core Intel Xeon (Skylake, IBRS) CPU (2.10 GHz) and 5 GB of RAM. On average, the call graph analysis and path construction take 22 minutes for the entire system, where the size of our studied systems varies from 79k to 1.2 million source lines of code. Note that the call graph analysis only needs to be done once, since, in practice, we can incrementally update the call graph based on the code changes in each commit. Therefore, we believe that the additional call graph analysis and path construction time is reasonably acceptable and would not affect the usability of the approach.

---

#### Algorithm 1. Execution Paths Re-Construction Algorithm

---

**Input:** Extracted Logs  
**Output:** Execution Paths

```

1: procedure FINDEXECUTIONPATH(logs)
2:   initialise executionPaths
3:   for  $i=1; i < \text{logs.length}$  do
4:     currentLog = logs.atPosition( $i$ )
5:     nextLog = logs.atPosition( $i+1$ )
6:     paths = findPathsBetween(currentLog, nextLog)
7:     executionPaths.add(paths)
8:   end for
9:   return executionPaths
10: end procedure
```

---

Once the execution paths are re-constructed, we compute the path score for every file on the execution paths. We compute the path score as follows:

$$\text{PathScore}(f, br) = \beta \times N(\text{VSMScore}(f, br)). \quad (3)$$

Given a bug report  $br$ ,  $\text{VSMScore}(f, br)$  is the cosine similarity score between file  $f$  and the bug report, where  $N$  is

## Logs

```

2008-01-07 21:02:13 INFO org.apache.hadoop.mapred.ReduceTask: task_r_1 done copying task_m_0
2008-01-07 21:02:13 INFO org.apache.hadoop.mapred.ReduceTask: task_r_1 Copying task_m_1
...
2008-01-07 21:02:13 WARN org.apache.hadoop.mapred.ReduceTask: java.lang.OutOfMemoryError: Java heap space
  at org.apache.hadoop.io.SequenceFile$Reader.init (SequenceFile.java:1345)
  at org.apache.hadoop.mapred.ReduceTask.run (ReduceTask.java:1311)
2008-01-07 21:02:31 ERROR org.apache.hadoop.mapred.ReduceTask: java.lang.NullPointerException: Map output copy failure
  at org.apache.hadoop.fs.InMemoryFileSystem.close (InMemoryFileSystem.java:378)
  at org.apache.hadoop.fs.FileSystem.getLength (FileSystem.java:449)
  at org.apache.hadoop.mapred.ReduceTask.run (ReduceTask.java:665)

```

Log Snippet #1			Stack Trace #1			Stack Trace #2		
Rank	File	Log Score	Rank	File	Log Score	Rank	File	Log Score
1	ReduceTask.java	0.10	1	SequenceFile.java	1.00	1	InMemoryFileSystem.java	1.00
			2	ReduceTask.java	0.50	2	FileSystem.java	0.50

Fig. 4. An example of the log score computation when there are log snippet and multiple stack traces.

the normalization function that normalizes  $VSMScore(f, br)$  to a value in the range of between 0 and 1, and  $\beta$  denotes the weight of  $VSMScore(f, br)$ , that is between 0 and 1. When a file appears on the path, the parameter  $\beta$  boosts the suspiciousness score to favor the files that were on the execution path. The buggy files may be one of the files that are on the execution path [13]. By introducing the path score, we are able to better distinguish the relevant files on the execution path from the less relevant files. In our study, we set  $\beta$  to 0.2 (we evaluate the effect of  $\beta$  in RQ3).

We explain the aforementioned path score computation in detail with Fig. 5 that serves as our running example. We derive the running example from a real bug report. We simplify the code for the ease of explanation and limit the call graph to a depth of one. In this example, our goal is to derive the execution path from the logs and compute the path score. This process of re-constructing the execution path is analogous to the sailing boat traveling back to the shore. The idea is that the running execution (the sailing boat) navigates through the logging statements (the beacons) in order to reach the potentially buggy classes (the shore). First, the logging statements are an analogy of “beacons”. The running execution finds and connects each of the logging statements in order to reconstruct the execution path. Starting from the reported logs, each log line is mapped to its corresponding logging statement by matching the static part of the logging statement. In our running example, by comparing each log line to the static part of the logging statements, we find that the log line “2019-01-07 21:02:13 INFO ReduceTask: task\_r\_1 initialized” is generated by the logging statement at line 3, and the log line “2019-01-07 21:02:13 INFO ReduceTask: runNewReducer called” is produced by the logging statement at line 12. Based on this information, we re-construct the execution path by connecting these two logging statements, and find the call path [2–6, 11–16] that contains both of the logging statements at line 3 and 12. Since the logging statement at line 18 is not executed nor recorded, the analysis excludes `runOldReducer()` in the execution path. Then, the classes that appear on the execution path are treated as the potentially buggy classes. The list of classes collected in our running example are: `JobConfiguration`, `Reducer`, and `Context`. Note that only the classes that are relevant to the project are collected. As these

classes might have data dependencies with the bug (or even contain the bug), we assign a PathScore to the files that contain these classes to boost their suspiciousness score based on Formula (3).

*Calculating the Final Suspiciousness Score.* To incorporate the vector space model, log analysis and path analysis into one combined component, we calculate the final suspiciousness score by summing up the normalized VSM score, log score, and path score (as shown in Formula (4)).

$$\begin{aligned}
 \text{FinalScore}(f, br) = & N(VSMScore(f, br)) \\
 & + \text{LogScore}(f, br) + \text{PathScore}(f, br).
 \end{aligned} \tag{4}$$

## 4 CASE STUDY SETUP

*Data Collection.* We evaluate the performance of our proposed bug localization approach on the bug reports, which contain logs (i.e., either log snippets, stack traces, or both), collected from eight open source systems. These systems are large in size, actively maintained, well-documented, and cover various domains ranging from big data processing to message brokers. For each system, we collect the bug reports as follows. The bug reports for all eight studied systems are available on the JIRA bug tracking repository [30]. Therefore, we implement a web crawler to collect bug reports from JIRA. We first select the bug reports that have the *resolution* status labeled as “Resolved” or “Fixed”, the *priority* field is marked as “Major”, “Critical”, or “Blocker”, and the *creation date* is 2010 or later. We download these bug reports in JSON format using the JIRA API [31]. Then, we examine the source code repository to further select the bug reports that have corresponding bug fixes by following prior studies [32], [33]. All the studied systems follow the convention to include the bug report identifier (e.g., HADOOP-1234) at the start of the commit messages (e.g., HADOOP – 1234.FixatypoinFileA.java) [34] and host the source code on Github. Therefore, we run the git command, `git log | grep bug_report_identifier[^\\d]`, to check if a bug report identifier exists in any commit message. If a bug report identifier appears in a commit message, then there is a bug fix for the bug and we identify the

## Logs

- ① 2019-01-07 21:02:13 INFO ReduceTask: task\_r\_1 initialized
- ② 2019-01-07 21:02:13 INFO ReduceTask: runNewReducer called
- ③ 2019-01-07 21:02:13 INFO ReduceTask: task\_r\_1 done

Code Snippets	Execution	Class boosted by path
1 class ReduceTask {	●	-
2 void run_task(String task_id, JobConfiguration job){	●	-
3 log.info(task_id + ' initialized'); //①	●	-
4 boolean useNewApi = job.getUseNewReducer();	●	JobConfiguration
5 if (useNewApi) {	●	-
6 runNewReducer();	●	-
7 } else {	○	-
8 runOldReducer();	○	-
9 }	○	-
10 log.info(id + ' done'); //③	●	-
11 void runNewReducer(){	●	-
12 log.info(' runNewReducer called'); //②	●	-
13 Reducer reducer = createReducer(job);	●	Reducer
14 Context context = createReduceContext();	●	Context
15 reducer.run(context);	●	-
16 }	●	-
17 void runOldReducer(){	○	-
18 log.info(' runOldReducer called');	○	-
19 ReduceValuesIter values = new ReduceValuesIter();	○	-
20 values.informReduceProcess();	○	-
21 }	○	-
22 }	○	-

Fig. 5. An example of the execution path analysis for path score computation.

commit as the bug fixing commit. Finally, to reduce noise, we exclude the bug reports in which no Java files were modified in the bug fixes. At the end of this process, we collected a total of 6,535 bug reports in the eight studied systems. After collecting the bug reports that have corresponding bug fixes, we further categorize the bug reports into *with logs* and *without logs*.

Table 1 shows an overview of the bug reports that we collected. We denote bug reports with logs as *BRWL* and bug reports without logs as *BRNL*. In total, there are 1,273 bug

reports with logs and 5,262 bug reports without logs. Although there are fewer bug reports with logs, the number is still non-negligible (around 20 percent of all bug reports). Thus, if we can leverage the embedded information in logs, we may better help improve bug localization.

*Metrics for Evaluating Pathidea.* To evaluate the effectiveness of our approach, we consider several commonly-used evaluation metrics for IR-based bug localization approaches [6], [12], [35]. First, we calculate the precision, recall, and F1-measure for the Top@N results (i.e., when examining the highest

TABLE 1  
An Overview of Our Studied Systems, Where BRWL Denotes Bug Reports With Logs,  
and BRNL Denotes Bug Reports Without Logs

System	LOC	# Bug reports	# BRWL	# BRNL
ActiveMQ	338k	594	86 (14%)	508 (86%)
Hadoop Common	190k	725	257 (35%)	468 (65%)
HDFS	285k	1,166	229 (20%)	937 (80%)
MapReduce	198k	575	166 (29%)	409 (71%)
YARN	548k	576	241 (42%)	335 (58%)
Hive	1.2M	2,231	195 (9%)	2,036 (91%)
Storm	275k	380	61 (16%)	319 (84%)
ZooKeeper	79k	288	38 (13%)	250 (87%)
<b>Total</b>	3.1M	6,535	1,273	5,262

ranked  $N$  files). Then, we calculate the mean average precision and mean reciprocal rank. Below, we briefly describe each of these metrics.

*Precision@N*. Given Top@ $N$ , the precision metric calculates the percentage of buggy files that are *correctly located* in the highest ranked  $N$  files. Precision@ $N$  is calculated as

$$\text{Precision}@N = \frac{\#\text{buggyfilesintop } N}{N}. \quad (5)$$

*Recall@N*. Given Top@ $N$ , the recall metric calculates the percentage of buggy files (out of all buggy files) that were located in the highest ranked  $N$  files. Recall@ $N$  is calculated as

$$\text{Recall}@N = \frac{\#\text{buggyfilesintop } N}{\#\text{totalbuggyfiles}}. \quad (6)$$

*F1@N*. F1-measure, also called F1 score is the weighted harmonic mean of precision and recall. This metric calculates the Top@ $N$  accuracy of the ranked results and offers a good trade-off between the precision and recall. F1 score is calculated as

$$F1@N = 2 \cdot \frac{\text{Precision}@N \cdot \text{Recall}@N}{\text{Precision}@N + \text{Recall}@N}. \quad (7)$$

*Mean Average Precision (MAP)*. MAP considers the ranks of all buggy files instead of only the first one. MAP is computed by taking the mean of the average precision across all bug reports. This metric is a commonly used in evaluating IR-based bug localization approach with ranked results. The average precision is calculated as

$$AP = \frac{\sum_{i=1}^m i / Pos(i)}{m}. \quad (8)$$

*Mean Reciprocal Rank (MRR)*. The reciprocal rank calculates the reciprocal of the position at which the first buggy file is found. MRR is the mean of reciprocal rank across all bug reports. Formula (9) calculates the mean reciprocal rank, where  $K$  denotes a set of bug reports, and  $rank_i$  is the rank of the first buggy file in the  $i$ th bug report.

$$MRR = \frac{1}{K} \sum_{i=1}^K \frac{1}{rank_i}. \quad (9)$$

## 5 CASE STUDY RESULTS

In this section, we discuss the results of our three research questions (RQs).

### 5.1 RQ1: Effectiveness of Pathidea Over State-of-the-Art Approaches

*Motivation*. Most prior research that uses information retrieval for bug localization (IRBL) only considers the textual information that is available in the bug report. Although logs contain textual information of the occurred events, logs can also be further mapped to source code and assist

developers with understanding the system execution during debugging and maintenance [13], [14]. Such system execution information may further help improve the performance of IRBL approaches. Therefore, in this RQ, we want to compare Pathidea with existing IRBL approaches.

*Approach*. As discussed in Section 3, for each bug report, we compute the final suspiciousness score for the files in the corresponding commit. We choose to use the commit that is prior to the bug fixing commit to avoid biases. We compare the performance of Pathidea with two baseline IRBL approaches. A recent study [35] shows that, among state-of-the-art IRBL approaches, BRTtracer achieves the best results in terms of MAP and MRR. In addition, similar to Pathidea, BRTtracer uses information in stack traces to improve bug localization results [16]. Hence, for the first baseline, we compare our approach with BRTtracer. For the second baseline, we compare Pathidea with the vanilla approach that uses the basic vector space model (VSM) for bug localization. We apply the approaches on all eight studied systems and compare their performance in terms of precision, recall, and F1-measure at the top 1, top 5, and top 10 ranked files. We also compare the MAP and MRR scores of the approaches. Finally, we use the Wilcoxon rank-sum test to investigate whether Pathidea achieves a statistically significant improvement over the baselines. We choose the Wilcoxon rank-sum test since it is a non-parametric test that does not have an assumption on the distribution of the underlying data [36].

*Result*. *Pathidea Significantly Outperforms BRTtracer and VSM in all Studied Systems With Respect to all Evaluation Metrics*. Table 2 compares the results between VSM, BRTtracer, and Pathidea. We find that for every studied system, VSM performs the worst among the three IRBL approaches. Thus, we focus our comparison between BRTtracer and Pathidea. The numbers in the parentheses show the percentage of the improvement of Pathidea over BRTtracer. Compared to BRTtracer, Pathidea achieves an average MAP of 35 percent across the studied systems, which is a 13 percent improvement over that of BRTtracer (i.e., 31 percent). Across the studied systems, the improvement in MAP varies from 8 to 24 percent. For MRR, Pathidea achieves an average of 43 percent across the studied systems, which is a 12 percent improvement over that of BRTtracer (i.e., 38 percent).

*Pathidea Achieves an Average Recall@10 of 50.3 percent, Which Shows That It Can Identify Half of the Buggy Files in a Relatively Short List*. Pathidea shows a large improvement in terms of Precision@ $N$  and Recall@ $N$ . Pathidea achieves, on average, 16, 12, and 11 percent improvement in Precision@1, 5, 10, respectively. For the average Recall@1, 5, 10, we see 20, 14, and 10 percent improvement, respectively. Regarding the F1-measures, Pathidea achieves an improvement between 15.5 and 31 percent for Top@1, and between 13.8 and 24.0 percent for Top@5. The average precision values indicate that 30.6 percent of the located files are actually buggy at Top@1, 13.6 percent at Top@5, and 8.2 percent at Top@10. The high average recall values indicate that Pathidea can locate 22.3 percent of all the buggy files at Top@1, 44.7 percent at Top@5, and 50.3 percent at Top@10. Note that since the number of buggy files is often small (i.e., the median number of buggy files is three in the studied systems), it is difficult to achieve a high precision in the ranked

**TABLE 2**  
Comparisons of Results Between VSM, BRTtracer and Pathidea

System	Approach	Top@1			Top@5			Top@10			MAP	MRR
		Precision(%)	Recall(%)	F1(%)	Precision(%)	Recall(%)	F1(%)	Precision(%)	Recall(%)	F1(%)		
ActiveMQ	VSM	7.0	3.6	4.7	4.7	14.9	7.1	3.4	20.2	5.8	0.11	0.15
	BRTtracer	23.3	16.6	19.4	10.9	33.5	16.5	6.9	40.1	11.7	0.28	0.34
	Pathidea	29.1 (+25%)	21.4 (+29%)	24.6 (+27%)	14.4 (+32%)	43.7 (+30%)	21.7 (+32%)	9.0 (+31%)	48.3 (+21%)	15.1 (+29%)	0.34 (+21%)	0.41 (+21%)
Hadoop Common	VSM	14.0	9.8	11.5	7.5	24.3	11.4	5.1	31.9	8.8	0.20	0.23
	BRTtracer	33.1	24.4	28.1	13.2	44.1	20.3	7.8	50.6	13.5	0.37	0.44
	Pathidea	36.2 (+9%)	27.2 (+11%)	31.0 (+10%)	13.9 (+6%)	46.9 (+7%)	21.5 (+6%)	8.2 (+5%)	53.4 (+6%)	14.2 (+5%)	0.40 (+8%)	0.47 (+7%)
HDFS	VSM	16.2	10.6	12.8	10.1	30.3	15.2	7.1	38.9	12.0	0.23	0.29
	BRTtracer	25.8	19.4	22.1	14.3	44.8	21.7	9.1	53.2	15.6	0.35	0.41
	Pathidea	31.9 (+24%)	23.8 (+23%)	27.2 (+23%)	15.8 (+10%)	50.4 (+12%)	24.0 (+10%)	9.9 (+8%)	57.7 (+9%)	16.9 (+8%)	0.40 (+14%)	0.46 (+12%)
MapReduce	VSM	13.3	9.1	10.8	6.5	21.4	10.0	4.2	27.7	7.3	0.17	0.21
	BRTtracer	18.7	13.8	15.9	10.7	38.4	16.8	6.4	44.4	11.2	0.27	0.32
	Pathidea	22.3 (+19%)	17.1 (+24%)	19.4 (+22%)	11.2 (+4%)	41.1 (+7%)	17.6 (+5%)	6.6 (+3%)	46.5 (+5%)	11.5 (+3%)	0.30 (+11%)	0.35 (+9%)
YARN	VSM	14.9	9.7	11.7	8.4	26.1	12.7	5.6	34.2	9.6	0.20	0.25
	BRTtracer	27.0	18.9	22.2	13.7	45.0	21.0	8.0	52.1	13.9	0.34	0.42
	Pathidea	35.3 (+31%)	26.0 (+38%)	30.0 (+35%)	15.4 (+12%)	51.9 (+15%)	23.7 (+13%)	8.7 (+8%)	56.8 (+9%)	15.0 (+8%)	0.41 (+21%)	0.48 (+14%)
Hive	VSM	9.7	5.2	6.8	7.2	17.7	10.2	5.4	26.9	9.0	0.15	0.20
	BRTtracer	37.4	23.9	29.2	13.9	40.3	20.7	7.8	44.4	13.3	0.35	0.46
	Pathidea	37.4 (+0%)	24.1 (+1%)	29.3 (+0%)	15.5 (+11%)	47.5 (+18%)	23.4 (+13%)	8.8 (+12%)	53.1 (+20%)	15.1 (+13%)	0.38 (+9%)	0.50 (+9%)
Storm	VSM	18.0	11.7	14.2	9.8	28.2	14.6	5.9	33.8	10.0	0.22	0.28
	BRTtracer	32.8	22.5	26.7	12.5	40.9	19.1	7.5	47.6	13.0	0.33	0.42
	Pathidea	34.4 (+5%)	25.0 (+11%)	29.0 (+8%)	14.1 (+13%)	45.7 (+12%)	21.5 (+13%)	8.2 (+9%)	50.5 (+6%)	14.1 (+8%)	0.37 (+12%)	0.45 (+5%)
ZooKeeper	VSM	5.3	2.8	3.7	3.2	9.0	4.7	2.9	14.9	4.8	0.10	0.12
	BRTtracer	13.2	9.4	11.0	7.9	26.3	12.1	5.0	32.0	8.6	0.21	0.24
	Pathidea	18.4 (+40%)	13.4 (+42%)	15.5 (+41%)	8.9 (+13%)	30.2 (+15%)	13.8 (+14%)	5.8 (+16%)	36.1 (+13%)	10.0 (+15%)	0.25 (+19%)	0.29 (+21%)
Average across studied systems	BRTtracer	26.4	18.6	21.8	12.1	39.2	18.5	7.3	45.6	12.6	0.31	0.38
	Pathidea	30.6 (+16%)	22.3 (+20%)	25.8 (+18%)	13.6 (+12%)	44.7 (+14%)	20.9 (+13%)	8.2 (+11%)	50.3 (+10%)	14.0 (+11%)	0.35 (+13%)	0.43 (+13%)

For each metric, we calculate the percentage of improvements that Pathidea achieves over BRTtracer.

results. However, our approach is able to achieve a relatively high recall within a small  $N$ . Hence, by only investigating a small number of files, developers may identify around half of the buggy files. We also use the Wilcoxon rank-sum test [37] to examine whether the improvements of Pathidea over BRTtracer are statistically significant. Our results show that the improvements are statistically significant in terms of MAP, MRR, recall, and precision values ( $p$ -value < 0.05).

Across the studied systems, Pathidea achieves an improvement that varies from 8 to 21 and 5 to 21 percent over BRTtracer in terms of MAP and MRR across the studied systems, respectively. We also find that both Pathidea and BRTtracer outperform the vanilla VSM in identifying buggy files. Moreover, Pathidea can identify buggy files with an average Recall@10 of 50.3 percent.

## 5.2 RQ2: Effectiveness of Path Analysis

**Motivation.** Previous studies [16], [38] leveraged logs to improve the ranking of the potential buggy files for further investigation. However, these approaches either directly consider logs (i.e., stack traces) as plain text, or only retrieve the files that appear directly in logs. In practice, developers not only examine the logs, but they also leverage the logs to re-construct the run-time execution paths of the system for debugging [13], [14]. Such path information may be helpful to not only Pathidea but also other IRBL approaches. Therefore, in this RQ, we study the effect of the path analysis on the performance of existing IRBL approaches.

**Approach.** Our goal is to study how much additional improvement can path analysis provide to IRBL approaches.

Thus, we first examine the effectiveness of Pathidea with and without path analysis. Then, we further study if path analysis can help improve existing IRBL approaches. In particular, we apply path analysis to BRTtracer, because it is shown to have one of the highest MAP and MRR among the IRBL approaches [35]. Moreover, BRTtracer leverages logs for bug localization (i.e., the class names that are recorded in stack traces), so we can study if path analysis provides additional information to BRTtracer’s log analysis. We also use the Wilcoxon signed-rank test [37] to investigate whether our path analysis provides a statistically significant improvement to these two IRBL approaches.

**Result. Considering Path Analysis Improves the Overall Effectiveness of Pathidea by up to 20 percent in Terms of the Evaluation Metrics.** Table 3 compares the results of Pathidea when considering different components. We show the evaluation metrics when different components are considered and evaluate the improvement over the VSM baseline. We focus on evaluating the effectiveness of Pathidea with and without path analysis. Specifically, when considering path analysis (i.e., VSM + log + path), Pathidea has an improvement of MAP that varies from 4 to 20 percent over the ones without path analysis (i.e., VSM + log) across the studied systems. The improvement of MRR varies from 4 to 17 percent across the studied systems. We also observe an average improvement of 14, 4, and 4 percent on Precision@1, 5, 10, respectively. For the average recall values, the improvements are 23, 6, and 3 percent for Recall@1, 5, 10, respectively. The Wilcoxon signed-rank test also shows that, for all the studied systems, the improvements are statistically significant for Recall@1, Recall@5, Precision@1, Precision@5, and F1@1 ( $p$ -value < 0.05). Our finding shows that the path analysis is able to help promote the ranking of the buggy files in the result.

**TABLE 3**  
Comparisons of Pathidea's Results When Considering Different Components

System	Approach	Top@1			Top@5			Top@10			MAP	MRR	
		Precision(%)	Recall(%)	F1(%)	Precision(%)	Recall(%)	F1(%)	Precision(%)	Recall(%)	F1(%)			
ActiveMQ	VSM	7.0	3.6	4.7	4.7	14.9	7.1	3.4	20.2	5.8	0.11	0.15	
	VSM + log	26.7 (+283%)	18.6 (+416%)	21.9 (+362%)	12.8 (+175%)	39.1 (+162%)	19.3 (+172%)	8.7 (+159%)	47.1 (+133%)	14.7 (+155%)	0.32 (+191%)	0.39 (+160%)	
	VSM + log + path	29.1 (+317%)	21.4 (+495%)	24.6 (+419%)	14.4 (+210%)	43.7 (+193%)	21.7 (+206%)	9.0 (+166%)	48.3 (+140%)	15.1 (+161%)	0.34 (+209%)	0.41 (+173%)	
Hadoop Common	VSM	14.0	9.8	11.5	7.5	24.3	11.4	5.1	31.9	8.8	0.20	0.23	
	VSM + log	35.4 (+153%)	25.9 (+165%)	29.9 (+160%)	13.8 (+84%)	46.2 (+90%)	21.2 (+86%)	8.1 (+59%)	52.6 (+65%)	14.0 (+60%)	0.39 (+95%)	0.46 (+100%)	
	VSM + log + path	36.2 (+158%)	27.2 (+177%)	31.0 (+169%)	13.9 (+86%)	46.9 (+93%)	21.5 (+88%)	8.2 (+60%)	53.4 (+68%)	14.2 (+61%)	0.40 (+100%)	0.47 (+104%)	
HDFS	VSM	16.2	10.6	12.8	10.1	30.3	15.2	7.1	38.9	12.0	0.23	0.29	
	VSM + log	30.1 (+86%)	22.3 (+110%)	25.7 (+100%)	15.8 (+56%)	50.1 (+66%)	24.0 (+58%)	9.8 (+39%)	57.4 (+47%)	16.8 (+40%)	0.38 (+65%)	0.45 (+55%)	
	VSM + log + path	31.9 (+97%)	23.8 (+123%)	27.2 (+112%)	15.8 (+56%)	50.4 (+67%)	24.0 (+58%)	9.9 (+40%)	57.7 (+48%)	16.9 (+41%)	0.40 (+74%)	0.46 (+59%)	
MapReduce	VSM	13.3	9.1	10.8	6.5	21.4	10.0	4.2	27.7	7.3	0.17	0.21	
	VSM + log	20.5 (+55%)	15.6 (+71%)	17.7 (+64%)	11.1 (+70%)	40.5 (+89%)	17.4 (+74%)	6.5 (+54%)	45.5 (+65%)	11.4 (+56%)	0.28 (+65%)	0.33 (+57%)	
	VSM + log + path	22.3 (+68%)	17.1 (+88%)	19.4 (+79%)	11.2 (+72%)	41.1 (+92%)	17.6 (+76%)	6.6 (+56%)	46.5 (+68%)	11.5 (+57%)	0.30 (+76%)	0.35 (+67%)	
YARN	VSM	14.9	9.7	11.7	8.4	26.1	12.7	5.6	34.2	9.6	0.20	0.25	
	VSM + log	33.2 (+122%)	23.5 (+142%)	27.5 (+134%)	14.9 (+78%)	50.4 (+93%)	23.0 (+82%)	8.6 (+54%)	56.5 (+65%)	14.9 (+56%)	0.39 (+95%)	0.47 (+88%)	
	VSM + log + path	35.3 (+136%)	26.0 (+169%)	30.0 (+155%)	15.4 (+83%)	51.9 (+99%)	23.7 (+87%)	8.7 (+56%)	56.8 (+66%)	15.0 (+57%)	0.40 (+100%)	0.48 (+92%)	
Hive	VSM	9.7	5.2	5.7	6.8	7.2	17.7	10.2	5.4	26.9	9.0	0.15	0.20
	VSM + log	36.9 (+279%)	22.7 (+336%)	28.1 (+314%)	15.3 (+113%)	46.7 (+164%)	23.0 (+125%)	8.6 (+60%)	52.0 (+93%)	14.8 (+65%)	0.37 (+147%)	0.49 (+145%)	
	VSM + log + path	37.4 (+284%)	24.1 (+362%)	29.3 (+331%)	15.5 (+116%)	47.5 (+168%)	23.4 (+129%)	8.8 (+64%)	53.1 (+98%)	15.1 (+69%)	0.38 (+153%)	0.50 (+150%)	
Storm	VSM	18.0	11.7	14.2	9.8	28.2	14.6	5.9	33.8	10.0	0.22	0.28	
	VSM + log	32.8 (+82%)	23.3 (+99%)	27.3 (+92%)	14.4 (+47%)	47.3 (+68%)	22.1 (+52%)	8.2 (+39%)	50.5 (+49%)	14.1 (+40%)	0.36 (+64%)	0.44 (+57%)	
	VSM + log + path	34.4 (+91%)	25.0 (+113%)	29.0 (+104%)	14.1 (+43%)	45.7 (+62%)	21.5 (+48%)	8.2 (+39%)	50.5 (+49%)	14.1 (+40%)	0.37 (+68%)	0.45 (+61%)	
ZooKeeper	VSM	5.3	2.8	3.7	3.2	9.0	4.7	2.9	14.9	4.8	0.10	0.12	
	VSM + log	15.8 (+200%)	12.0 (+325%)	13.7 (+271%)	8.4 (+167%)	27.6 (+208%)	12.9 (+176%)	5.3 (+82%)	33.3 (+124%)	9.1 (+88%)	0.23 (+130%)	0.27 (+125%)	
	VSM + log + path	18.4 (+250%)	13.4 (+371%)	15.4 (+320%)	8.9 (+183%)	30.2 (+237%)	13.8 (+196%)	5.8 (+100%)	36.1 (+143%)	10.0 (+106%)	0.25 (+150%)	0.29 (+142%)	
Average across studied systems	VSM	12.3	7.8	9.5	7.2	21.5	10.7	5.0	28.6	8.4	0.17	0.22	
	VSM + log	28.9 (+135%)	20.5 (+162%)	24.0 (+152%)	13.3 (+86%)	43.5 (+102%)	20.4 (+90%)	8.0 (+61%)	49.4 (+73%)	13.7 (+63%)	0.34 (+97%)	0.41 (+91%)	
	VSM + log + path	30.6 (+149%)	22.2 (+185%)	25.8 (+170%)	13.6 (+90%)	44.7 (+108%)	20.9 (+95%)	8.2 (+65%)	50.3 (+76%)	14.0 (+66%)	0.35 (+106%)	0.43 (+97%)	

For each added component, we show the percentage of improvements over the VSM baseline.

*When Applying Path Analysis on an Existing IRBL Approach (i.e., BRTracer), There is a 10 and 8 percent Improvement in Terms of MAP and MRR, Respectively.* Table 4 compares the results of BRTracer with and without considering path analysis. We observe that, when considering path analysis, the MAP and MRR of BRTracer receive a 10 and 7 percent improvement, respectively. We also observe an improvement on the precision, recall, and F1, and most notably on Precision@1 and Recall@1. Specifically, the path analysis improves the average precision values by 12, 5, 5 percent for Precision@1, 5, 10, respectively. The improvement on the average recall values are 14, 5, 4 percent for Recall@1, 5, 10, respectively. Our finding shows that the path analysis may provide the largest improvement to

BRTracer especially when  $N$  equals to 1. The Wilcoxon signed-rank test shows that, for all studied systems, the improvements are statistically significant for precision, recall, and F1. Compared to Table 3, we find that the path analysis provides more improvement to BRTracer compared to Pathidea. For example, in YARN, adding the path analysis to BRTracer improves all the evaluation metrics when Top@1 (from 28 to 30 percent), where as the improvement in Pathidea is only 6 to 11 percent. Our finding shows that the path analysis can provide additional information to not only Pathidea, but also other IRBL approaches (e.g., BRTracer). Future studies may consider integrating the path information to improve bug localization performance.

**TABLE 4**  
Comparisons of BRTracer's Results With and Without Path Analysis

System	Approach	Top@1			Top@5			Top@10			MAP	MRR
		Precision(%)	Recall(%)	F1(%)	Precision(%)	Recall(%)	F1(%)	Precision(%)	Recall(%)	F1(%)		
ActiveMQ	BRTracer	23.3	16.6	19.4	10.9	33.5	16.5	6.9	40.1	11.7	0.28	0.34
	BRTracer + path	26.7 (+15%)	19.1 (+15%)	22.3 (+15%)	12.1 (+11%)	36.7 (+10%)	18.2 (+10%)	7.2 (+5%)	42.0 (+5%)	12.3 (+5%)	0.30 (+7%)	0.37 (+8%)
Hadoop Common	BRTracer	33.1	24.4	28.1	13.2	44.1	20.3	7.8	50.6	13.5	0.37	0.44
	BRTracer + path	35.0 (+6%)	26.4 (+8%)	30.1 (+7%)	13.9 (+6%)	46.6 (+6%)	21.4 (+6%)	8.1 (+4%)	52.3 (+3%)	14.0 (+4%)	0.39 (+5%)	0.46 (+5%)
HDFS	BRTracer	25.8	19.4	22.1	14.3	44.8	21.7	9.1	53.2	15.6	0.35	0.41
	BRTracer + path	30.1 (+17%)	22.8 (+18%)	25.9 (+17%)	15.1 (+5%)	47.5 (+6%)	22.9 (+6%)	9.7 (+6%)	56.3 (+6%)	16.5 (+6%)	0.38 (+9%)	0.45 (+10%)
MapReduce	BRTracer	18.7	13.8	15.9	10.7	38.4	16.8	6.4	44.4	11.2	0.27	0.32
	BRTracer + path	21.1 (+13%)	16.3 (+18%)	18.4 (+16%)	10.7 (0%)	39.5 (+3%)	16.9 (+1%)	6.4 (0%)	44.8 (+1%)	11.2 (0%)	0.29 (+7%)	0.33 (+3%)
YARN	BRTracer	27.0	18.9	22.2	13.7	45.0	21.0	8.0	52.1	13.9	0.34	0.42
	BRTracer + path	34.4 (+28%)	24.9 (+32%)	28.9 (+30%)	14.9 (+8%)	49.5 (+10%)	22.9 (+9%)	8.4 (+5%)	54.7 (+5%)	14.6 (+5%)	0.39 (+15%)	0.48 (+14%)
Hive	BRTracer	37.4	23.9	29.2	13.9	40.3	20.7	7.8	44.4	13.3	0.35	0.46
	BRTracer + path	37.4 (0%)	24.3 (+2%)	29.5 (+1%)	14.3 (+2%)	42.0 (+4%)	21.3 (+3%)	7.9 (+1%)	45.6 (+3%)	13.5 (+2%)	0.36 (+3%)	0.47 (+2%)
Storm	BRTracer	32.8	22.5	26.7	12.5	40.9	19.1	7.5	47.6	13.0	0.33	0.43
	BRTracer + path	32.8 (0%)	22.5 (0%)	26.7 (0%)	12.5 (0%)	40.9 (0%)	19.1 (0%)	7.4 (-2%)	47.1 (-1%)	12.8 (-2%)	0.33 (0%)	0.43 (0%)
ZooKeeper	BRTracer	13.2	9.4	11.0	7.9	26.3	12.1	5.0	32.0	8.6	0.21	0.24
	BRTracer + path	18.4 (+40%)	13.4 (+42%)	15.5 (+41%)	8.4 (+7%)	27.6 (+5%)	12.9 (+6%)	6.1 (+21%)	36.3 (+14%)	10.4 (+20%)	0.25 (+19%)	0.29 (+21%)
Average across studied systems	BRTracer	26.4	18.6	21.8	12.1	39.2	18.5	7.3	45.6	12.6	0.31	0.38
	BRTracer + path	29.5 (+12%)	21.2 (+14%)	24.7 (+13%)	12.7 (+5%)	41.3 (+5%)	19.5 (+5%)	7.7 (+5%)	47.4 (+4%)	13.2 (+5%)	0.34 (+10%)	0.41 (+8%)

For each added component, we show the percentage improvement over the original BRTracer.

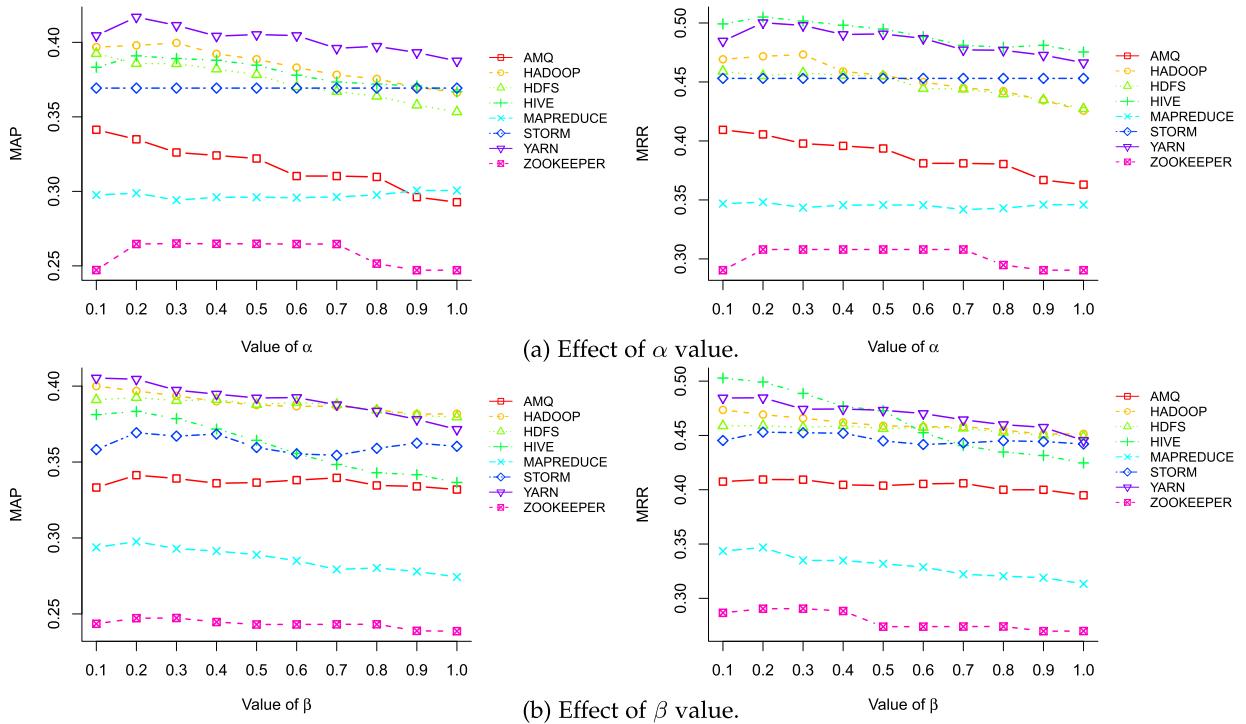


Fig. 6. Effect of  $\alpha$  and  $\beta$  on Pathidea in terms of MAP and MRR.

The re-constructed execution paths can complement BRTtracer by providing a 10 and 8 percent improvement in MAP and MRR, respectively. We also find that Pathidea provides an average of 16 percent improvement over BRTtracer on Precision@1. Future IRBL research may consider combining information in the source code (e.g., execution paths re-constructed from logs) to further improve bug localization performance.

### 5.3 RQ3: Parameter Sensitivity of Pathidea

*Motivation:* As mentioned in Section 3, Pathidea uses two parameters  $\alpha$  and  $\beta$  to calculate the final suspiciousness score. In each system, there may be some system-specific characteristics (e.g., lexical similarity, semantic redundancy of source code, and log density) that make the contribution of one component more important than others. For instance, if a system allocates a significant amount of effort on improving and maintaining logging statements for debugging, then the attached logs in the bug reports may contain more information compared to other systems. In such case, we may want to attribute more weight to the  $\alpha$  parameter which is related to the logging statements. Therefore, in this RQ, we want to further investigate the sensitivity of the parameters on the overall effectiveness of Pathidea.

*Approach.* The parameter  $\alpha$  serves to attribute a suspiciousness score to each file that appears in the logs (i.e., calculating *LogScore* in Equation (2)). The parameter  $\beta$  serves as a magnifier that adjusts the weight of *VSMScore* to favor the files on the re-constructed execution paths (i.e., calculating *PathScore* in Equation (3)). To understand the effect of these parameters on Pathidea, we perform a sensitivity analysis on the parameters separately by changing the

values between 0.1 to 1.0, with an interval of 0.1, to quantify their effects in terms of the MAP and MRR values.

*Result.* Overall, the MAP and MRR Values Reach the Highest When  $\alpha$  and  $\beta$  are in the Range of 0.1 and 0.2. However, We Also Observe Some Variations Among the Studied Systems. Fig. 6a shows the effectiveness of Pathidea when varying the parameter  $\alpha$ . We observe a relatively stable impact of  $\alpha$  across the studied systems. For Hadoop Common, Hive, MapReduce, YARN and ZooKeeper, when  $\alpha$  increases from 0.1 to 0.2, we observe an improvement in terms of the MAP and MRR values. From 0.3 to 0.7, the MAP and MRR values remain relatively stable. Starting from 0.8 to 1.0, the MAP and MRR values decrease. The effect of  $\alpha$  on ActiveMQ is different from the other systems. In ActiveMQ, the values of MAP and MRR decrease when the parameter  $\alpha$  value increases. For MapReduce and Storm, the MAP and MRR values remain stable no matter how the parameter  $\alpha$  varies. Fig. 6b shows the effectiveness of Pathidea when varying the parameter  $\beta$ . Almost all systems achieve the highest MAP and MRR values when  $\beta$  is between 0.1 and 0.2. The further increase of  $\beta$  does not improve the MAP and MRR values for AMQ, MapReduce and Storm. In these three systems, the values of MAP and MRR decrease when  $\beta$  varies from 0.3 to 1.0. In summary, practitioners and future studies may consider setting the value of  $\alpha$  and  $\beta$  between the range of 0.1 and 0.2 when applying Pathidea.

In general, Pathidea has the highest MRR and MAP values when the values of  $\alpha$  and  $\beta$  are in the range of 0.1 and 0.2. Practitioners and future studies may consider choosing these values when integrating or applying Pathidea.

## 6 DISCUSSION

*Studying the Effectiveness of the Path Analysis When Added to BRTtracer.* In Section 5.2 (RQ2), we observe that Storm and

Hive experience the least improvement when applying the path analysis on BRTracer (i.e., Table 4). In Hive, the improvements are 3 percent for MAP and 2 percent for MRR; and in Storm, the improvements for MAP and MRR are both 0 percent. After some investigation, we find that there is one possible factor that may be correlated with the relatively lower improvement when the path analysis is applied to BRTracer. We find that, among all the studied systems, Storm and Hive have the largest percentage of bug reports that contain only stack traces. 86.5 and 69.8 percent of the bug reports with logs contain only stack traces but no log snippets in Hive and Storm, respectively. On the other hand, there is only an average of 50.7 percent of such bug reports in other studied systems. A prior study [39] finds that many of the bug fixing locations may not be directly related to the reported stack traces. The stack traces may only show the symptom of the bug (e.g., `NullPointerException`), but the actual bug may manifest in a file that was called earlier during the execution (e.g., developers did not check the returned value in earlier method calls, which eventually results in a `NullPointerException`). Therefore, it may be possible that some buggy files are not related to the files on the re-constructed paths. In other words, path analysis may be less effective on the bug reports that only have stack traces than the ones that have both stack traces and log snippets. Note that the path analysis still has a relatively larger improvement for Storm and Hive when added to Pathidea. The possible reason may be that Pathidea has a different log score computation than BRTracer, which may provide larger improvements to the bug reports that contain log snippets. Future studies are needed to further understand the effect of log quality on the bug localization performance.

Another possible factor that affects the effectiveness of path analysis is the log density of a system. The log density is calculated by the ratio between thousands of lines of logging code and LOC ( $\frac{\text{linesofloggingcode}}{\text{thousandsoflinesofcode}}$ ). Intuitively, less noise would be introduced when re-constructing an execution paths with the reported log snippets if the log density is higher. To test our assumption, we calculate the log density of all the studied systems (Table 5). For instance, when considering path analysis on either Pathidea or BRTracer, we observe a substantial improvement in ZooKeeper (i.e., it has the highest log density among all studied systems) under all metrics. Specifically, when the path analysis is applied on BRTracer, the metrics of *Precision@1*, *Recall@1* and *F1@1* increase by 40, 42 and 41 percent, respectively. The improvement is 19 percent for MAP, and 21 percent for MRR. ZooKeeper has the highest log density (Table 5). There is one line of logging code for every 33 lines of code. In Hive, where its log density is the lowest among the studied systems, we observe that the improvements are relatively small. Future studies are needed to examine the effect of log density on the effectiveness of the re-constructed execution paths in bug localization.

*Effectiveness of Segmentation.* Table 6 shows the effectiveness of segmentation at different segment sizes. We evaluate the effectiveness based on *Precision@1*, *Precision@5*, *Precision@10*, MAP and MRR. We observe that, for most of

TABLE 5  
Log Density Across Studied Systems, Where LOC is  
Referred to as Lines of Code, and LOLC is  
Referred as Lines of Logging Code

System	LOC	LOLC	LOLC per every thousand LOC
<b>ActiveMQ</b> (5.15.13)	337,533	8,055	24
<b>Hadoop</b>	189,744	2,471	13
<b>Common</b> (2.7.6)	285,071	5,971	21
<b>HDFS</b> (2.7.0)	197,996	3,279	17
<b>MapReduce</b> (3.1.4)	548,043	6,854	13
<b>YARN</b> (3.1.2)	1,180,562	9,918	8
<b>Hive</b> (2.7.0)	274,860	5,620	20
<b>Storm</b> (2.2.0)	78,684	2,518	32
<b>ZooKeeper</b> (3.6.0)	3,092,493	44,686	19

Note that we exclude code comments and empty lines.

the studied systems (i.e., ActiveMQ, Hadoop, Hive, Storm and ZooKeeper), 400 is the segment size that yields the most effective metrics, while the most effective segment size is 600 for HDFS and MapReduce, and 800 for Yarn. Although the optimal segmentation size is different for each studied system, we observe a trend where smaller segmentation sizes (e.g., around or below 800) yields better localization results. Future studies and practitioners may consider using smaller segmentation sizes (e.g., 800 or below) when adopting the technique.

*Parameters Settings.* Throughout our experiment, we have tuned these parameters to evaluate the effectiveness of our approach at different thresholds. Our experiment indicates that, for most of the studied systems, the MAP and MRR values achieve the best localization results when  $\alpha$  and  $\beta$  are in the range of 0.1 and 0.2, and when the segment size is between 400 to 800. Although the optimized parameter setting can vary from system to system, some system characteristics may be related to the most effective parameter values. We observe a trend that smaller segmentation sizes (e.g., 800 or below) yield the best localization results, especially for smaller systems. For instance, Zookeeper, which is the smallest among all studied systems, has the best localization results when the segment size is 400. In contrast, Hive, which is the largest among all studied systems, has the best localization results when the segment size is 1,000. Therefore, future studies and practitioners may consider starting with a smaller segment size for smaller systems and gradually increase the segment size to find the optimal value. We also observe that, when the logging statements are too far from each other (i.e., low log density), there may be more noises when we reconstruct the execution path. The  $\alpha$  value decides on how the logged classes are boosted. In general, we observe that the systems with a higher log density are more sensitive to the change of  $\alpha$  value (i.e., the magnifier parameter given to the files found in the reported logs). In Fig. 6a, the two systems with the highest log density, AMQ and ZooKeeper, have a large variation in their effectiveness as the  $\alpha$  value varies and increases. Therefore, we suggest that

**TABLE 6**  
Effectiveness of Segmentation at Different Segment Sizes, Where *Size* Column Refers to Segment Size

System	Size	Precision@1	Precision@5	Precision@10	MAP	MRR
<b>ActiveMQ</b>	400	10.5	4.9	4.1	0.13	0.18
	600	8.1	5.3	3.4	0.12	0.16
	800	7.0	4.7	3.4	0.11	0.15
	1000	4.7	4.4	3.3	0.09	0.12
	1200	4.7	4.7	3.4	0.09	0.12
<b>Hadoop</b>	400	17.1	9.3	6.1	0.22	0.28
	600	16.0	8.4	5.6	0.21	0.26
	800	14.0	7.5	5.1	0.20	0.23
	1000	13.6	7.1	5.1	0.19	0.22
	1200	12.5	6.6	4.7	0.18	0.20
<b>HDFS</b>	400	18.8	9.9	6.7	0.24	0.30
	600	21.4	10.7	6.9	0.26	0.32
	800	16.2	10.1	7.1	0.23	0.29
	1000	17.5	10.0	6.6	0.23	0.29
	1200	17.0	9.3	6.4	0.22	0.28
<b>MapReduce</b>	400	12.0	7.1	5.2	0.17	0.21
	600	12.7	6.9	5.1	0.18	0.21
	800	13.3	6.5	4.3	0.17	0.21
	1000	10.8	5.7	3.9	0.16	0.19
	1200	13.9	5.7	3.9	0.16	0.20
<b>YARN</b>	400	14.5	8.1	5.6	0.20	0.25
	600	12.4	8.3	5.8	0.18	0.23
	800	14.9	8.4	5.6	0.20	0.25
	1000	14.1	8.2	5.5	0.20	0.24
	1200	13.3	8.5	5.6	0.19	0.23
<b>Hive</b>	400	11.7	7.1	5.6	0.15	0.20
	600	10.2	7.2	5.5	0.14	0.20
	800	10.8	7.3	5.4	0.15	0.20
	1000	13.8	7.3	5.0	0.14	0.22
	1200	12.8	7.1	4.5	0.14	0.21
<b>Storm</b>	400	21.3	10.2	6.9	0.25	0.31
	600	19.7	9.2	6.4	0.22	0.29
	800	18.0	10.2	6.1	0.22	0.28
	1000	21.3	8.5	5.6	0.22	0.29
	1200	21.3	8.2	5.2	0.22	0.29
<b>ZooKeeper</b>	400	13.2	5.3	3.9	0.15	0.20
	600	7.9	4.7	2.9	0.12	0.15
	800	5.3	3.2	2.9	0.10	0.12
	1000	7.9	4.2	3.4	0.11	0.15
	1200	7.9	3.7	3.2	0.11	0.14

future studies and practitioners may want to start with a smaller  $\alpha$  when the log density of the system is larger.  $\beta$ , which serves as a magnifier for PathScore, decides on how the classes in the path are boosted. A higher  $\beta$  value attributes larger weight to the classes that are on the execution paths. For larger systems that have a low log density, such as Hive, we observe that the localization accuracy is the highest when the  $\beta$  value remains small. In Fig. 6b, we observe that both the MAP and MRR values for Hive fall drastically as the  $\beta$  value increases. This may be that larger systems with lower log density value will have longer execution paths, which leads to a considerable amount of classes boosted by the  $\beta$  parameter. The localization accuracy decreases when too many classes are boosted (i.e., more noise). Thus, we suggest that future studies and

practitioners may want to start with a smaller  $\beta$  when the system has a lower log density. In summary, we recommend future studies to set the initial parameter values small and increase them slowly (e.g., by 0.1) to find the optimal parameter values for the system.

## 7 THREATS TO VAIDILITY

*External Validity.* Threats to external validity relates to the generalizability of our findings. To reduce this threat, we conduct our case study on eight large-scale open source systems that vary in size and infrastructures (i.e., data warehouse, realtime computation system, distributed file system). These systems are actively maintained and widely used. Although all the systems are Java-based, our approach is not limited to Java

systems. We present our approach in a generic way that can easily be adapted to fit other programming languages. For uncovering the execution paths, another AST parser that fits the programming language should be used to replace Java-parser (e.g., *ast* module [40] for Python, and *cppast* library [41] for C++). Apart from execution paths, the mapping of the user-reported logs to the logging statements should be customized to fit the logging practice of the programming language. Future research is encouraged to be conducted on more bug reports from more systems written in different programming languages.

*Construct Validity.* Threats to construct validity refer to the suitability of the set of evaluation metrics that we use in this study. To reduce the threat, we use five evaluation metrics in our study: Recall@ $N$ , Precision@ $N$ , F1@ $N$ , MAP, and MRR. These metrics are commonly used in information retrieval and have been used to evaluate many prior bug localization techniques [12], [35], [42], [43]. We did not consider control flow analysis in our approach. In some cases, considering the control flow may provide more information. However, one challenge is that logs are relatively sparse in the code, so the accuracy of finer-grained control flow analysis will be low. Moreover, in our prior work [15], we investigated the benefits and challenges of analyzing logs in bug reports. We found that developers may have made some code changes (i.e., the version that the user reported the issue is an older version and the code has changed), and the logging statements might be removed throughout the source code evolution (i.e., the user reported logs can no longer be found in the source code). Therefore, to reduce some noises caused by code evolution and the sparseness of logs, we decided to design the approach by generating the call graph and conduct the analysis at the file level.

## 8 CONCLUSION

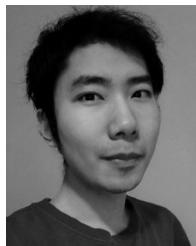
To help developers with debugging, researchers have proposed a series of information retrieval-based bug localization (IRBL) approaches. IRBL approaches aim to find the source code files that have the highest textual similarity with a given bug report for further investigation. However, in bug reports, in addition to the textual information describing the bug, reporters also often attach logs. Logs illustrate the system execution information when the bug occurs and can be mapped to the source code to re-construct the system execution paths. However, such information is not directly “visible” in bug reports and is not utilized by prior IRBL approaches. In this paper, we propose Pathidea, a IRBL approach that leverages logs in bug reports to reconstruct execution paths. Pathidea integrates the execution paths information to further improve the performance of bug localization. Our evaluation on eight open source systems shows that Pathidea can identify buggy files with high recall values (up to 51.9 and 57.7 percent for Top@5 and Top@10, respectively). Pathidea outperforms existing state-of-the-art IRBL approaches and achieves an average improvement that varies from 8 to 21 and 5 to 21 percent over BRTtracer in terms of MAP and MRR, respectively. In addition, our results also show that the re-constructed execution paths can complement existing IRBL approaches by providing a 10 and 8 percent improvement in terms of MAP

and MRR, respectively. Finally, we provide recommendations to practitioners on setting the parameter values in Pathidea. In short, our study highlights the benefit of integrating the system execution information, and future studies may consider leveraging such execution paths information when designing IRBL approaches.

## REFERENCES

- [1] T. D. LaToza and B. A. Myers, “Developers ask reachability questions,” in *Proc. ACM/IEEE Int. Conf. Softw. Eng.*, 2010, pp. 185–194.
- [2] T.-H. Chen, M. Nagappan, E. Shihab, and A. E. Hassan, “An empirical study of dormant bugs,” in *Proc. Working Conf. Mining Softw. Repositories*, 2014, pp. 82–91.
- [3] 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 *Proc. Int. Conf. Softw. Eng.*, 2012, pp. 14–24.
- [4] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, “Improving bug localization using structured information retrieval,” in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2013, pp. 345–355.
- [5] A. N. Lam, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, “Bug localization with combination of deep learning and information retrieval,” in *Proc. Int. Conf. Program Comprehension*, 2017, pp. 218–229.
- [6] S. Wang and D. Lo, “Amalgam+: Composing rich information sources for accurate bug localization,” *J. Softw. Evol. Process*, vol. 28, no. 10, pp. 921–942, 2016.
- [7] B. Liu, Lucia, S. Nejati, L. C. Briand, and T. Bruckmann, “Simulink fault localization: An iterative statistical debugging approach,” *Softw. Testing, Verification Rel.*, vol. 26, no. 6, pp. 431–459, 2016.
- [8] P. Loyola, K. Gajananan, and F. Satoh, “Bug localization by learning to rank and represent bug inducing changes,” in *Proc. ACM Int. Conf. Inf. Knowl. Manage.*, 2018, pp. 657–665.
- [9] T. Dao, L. Zhang, and N. Meng, “How does execution information help with information-retrieval based bug localization?” in *Proc. Int. Conf. Program Comprehension*, 2017, pp. 241–250.
- [10] R. Bhagwan, R. Kumar, C. S. Maddila, and A. A. Philip, “Orca: Differential bug localization in large-scale services,” in *Proc. USENIX Symp. Operating Syst. Des. Implementation*, 2018, pp. 493–509.
- [11] B. Sisman and A. C. Kak, “Incorporating version histories in information retrieval based bug localization,” in *Proc. IEEE Working Conf. Mining Softw. Repositories*, 2012, pp. 50–59.
- [12] M. Wen, R. Wu, and S.-C. Cheung, “Locus: Locating bugs from software changes,” in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2016, pp. 262–273.
- [13] D. Yuan, H. Mai, W. Xiong, L. Tan, Y. Zhou, and S. Pasupathy, “Sherlog: Error diagnosis by connecting clues from run-time logs,” in *Proc. Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2010, pp. 143–154.
- [14] Z. Li, T.-H. P. Chen, J. Yang, and W. Shang, “DLfinder: Characterizing and detecting duplicate logging code smells,” in *Proc. Int. Conf. Softw. Eng.*, 2019, pp. 152–163.
- [15] A. R. Chen, T.-H. P. Chen, and S. Wang, “Demystifying the challenges and benefits of analyzing user-reported logs in bug reports,” *Empirical Softw. Eng.*, vol. 26, 2020, Art. no. 8.
- [16] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei, “Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis,” in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2014, pp. 181–190.
- [17] B. Chen and Z. M. (Jack) Jiang, “Characterizing logging practices in java-based open source software projects – A replication study in apache software foundation,” *Empirical Softw. Eng.*, vol. 22, no. 1, pp. 330–374, Feb. 2017.
- [18] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, “What makes a good bug report?,” in *Proc. Int. Symp. Found. Softw. Eng.*, 2008, pp. 308–318.
- [19] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, “A survey on software fault localization,” *IEEE Trans. Softw. Eng.*, vol. 42, no. 8, pp. 707–740, Aug. 2016.
- [20] A. Zakari, S. P. Lee, R. Abreu, B. H. Ahmed, and R. A. Rasheed, “Multiple fault localization of software programs: A systematic literature review,” *Inf. Softw. Technol.*, vol. 124, 2020, Art. no. 106312.
- [21] S. Pearson *et al.*, “Evaluating and improving fault localization,” in *Proc. IEEE/ACM Int. Conf. Softw. Eng.*, 2017, pp. 609–620.

- [22] K. C. Youm, J. Ahn, and E. Lee, "Improved bug localization based on code change histories and bug reports," *Inf. Softw. Technol.*, vol. 82, pp. 177–192, 2017.
- [23] A. R. Chen, "An empirical study on leveraging logs for debugging production failures," in *Proc. IEEE/ACM Int. Conf. Softw. Eng. Companion Proc.*, 2019, pp. 126–128.
- [24] T.-H. Chen, S. W. Thomas, and A. E. Hassan, "A survey on the use of topic models when mining software repositories," *Empirical Softw. Eng.*, vol. 21, no. 5, pp. 1843–1919, 2016.
- [25] Oracle, "Java language keywords." Accessed: Feb. 01, 2020. [Online]. Available: [https://docs.oracle.com/javase/tutorial/java/nutsandbolts/\\_keywords.html](https://docs.oracle.com/javase/tutorial/java/nutsandbolts/_keywords.html)
- [26] NLTK, "NLTK corpora." Accessed: Feb. 01, 2020. [Online]. Available: [https://www.nltk.org/nltk\\_data/](https://www.nltk.org/nltk_data/)
- [27] B. Chen, J. Song, P. Xu, X. Hu, and Z. M. J. Jiang, "An automated approach to estimating code coverage measures via execution logs," in *Proc. ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2018, pp. 305–316.
- [28] T.-H. Chen *et al.*, "Analytics-driven load testing: An industrial experience report on load testing of large-scale systems," in *Proc. Int. Conf. Softw. Eng. Softw. Eng. Pract. Track*, 2017, pp. 243–252.
- [29] Accessed: Jul. 1, 2020. [Online]. Available: <https://javaparser.org/>
- [30] Apache, "Apache JIRA," 2020. Accessed: Feb. 01, 2020. [Online]. Available: <https://issues.apache.org/jira/>
- [31] Jira, "Jira REST APIs," 2020. Accessed: Feb. 01, 2020. [Online]. Available: <https://developer.atlassian.com/server/jira/platform/rest-apis/>
- [32] S. Kim, T. Zimmermann, K. Pan, and E. J. J. Whitehead, "Automatic identification of bug-introducing changes," in *Proc. Int. Conf. Automated Softw. Eng.*, 2006, pp. 81–90.
- [33] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?," vol. 30, pp. 1–5, 2005.
- [34] A. Ajisaka, "How to contribute to apache hadoop," Accessed: Feb. 01, 2020. <https://cwiki.apache.org/confluence/display/HADOOP/How+To+Contribute>
- [35] J. Lee, D. Kim, T. F. Bissyandé, W. Jung, and Y. Le Traon, "Bench4bl: Reproducibility study on the performance of IR-based bug localization," in *Proc. ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2018, pp. 61–72.
- [36] D. Moore, G. MacCabe, and B. Craig, *Introduction to the Practice of Statistics*. New York, NY, USA: W.H. Freeman Company, 2009.
- [37] F. Wilcoxon, "Individual comparisons by ranking methods," in *Breakthroughs in Statistics*. New York, NY, USA: Springer, 1992, pp. 196–202.
- [38] S. Davies and M. Roper, "Bug localisation through diverse sources of information," in *Proc. IEEE Int. Symp. Softw. Rel. Eng. Workshops*, pp. 126–131.
- [39] C. An Ran , "Studying and leveraging user-provided logs in bug reports for debugging assistance," M.A.Sc. Thesis, Dept. Comput. Sci. Softw. Eng., Concordia Univ., Montreal, QC, Canada, 2019.
- [40] ast — Abstract syntax trees. Accessed: Jul. 20, 2020. [Online]. Available: <https://docs.python.org/3/library/ast.html>
- [41] Library to parse and work with the C++ ast. Accessed: Jul. 20, 2020. [Online]. Available: <https://github.com/foonathan/cppast>
- [42] S. Wang and D. Lo, "Version history, similar report, and structure: Putting them together for improved bug localization," in *Proc. Int. Conf. Prog. Comprehension*, 2014, pp. 53–63.
- [43] S. Wang and D. Lo, "Amalgam+: Composing rich information sources for accurate bug localization," *J. Softw. Evol. Process*, vol. 28, no. 10, pp. 921–942, 2016.



**An Ran Chen** (Student Member, IEEE) received the bachelor's and master's degree in software engineering from Concordia University, where he is currently working toward the PhD degree in software engineering . He is currently a research assistant with Software PErfomance, Analysis, and Reliability (SPEAR) Lab. Before joining SPEAR, he was a software engineer with the Bank of Canada and McGill University IT Services. His research interests include program analysis, log analysis, mining software repositories, software testing, and debugging.



**Tse-Hsun Chen** (Member, IEEE) received the BSc degree from the University of British Columbia, and the MSc and PhD degrees from Queen's University. He was a software performance engineer with BlackBerry for more than four years. He developed tools, which were integrated into industrial practice for ensuring the quality of large-scale enterprise systems. He is currently an assistant professor with the Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada. He leads the Software PEfomance, Analysis, and Reliability (SPEAR) Lab, which focuses on conducting research on performance engineering, program analysis, log analysis, production debugging, and mining software repositories. His work is adopted by companies, such as BlackBerry and Ericsson, and has been published in flagship conferences and journals, such as ICSE, FSE, TSE, EMSE, and MSR. He is also a program committee member in international software engineering conferences, such as FSE, ASE, ICSME, SANER, and ICPC. He is also a regular reviewer for software engineering journals, such as JSS, EMSE, and TSE, and the guest editor for special issues in EMSE.



**Shaowei Wang** (Member, IEEE) received the BSc degree from Zhejiang University and the PhD degree from Singapore Management University. He is currently an assistant professor with the Department of Computer Science, University of Manitoba. His work has been published in flagship conferences and journals, such as ASE, ICMSE, TSE, and TOSEM. His research interests include software engineering, machine learning, data analytics for software engineering, automated debugging, and secure software development. He was one of four recipients of the 2018 Distinguished Reviewer Award from the Springer EMSE (SE's highest impact journal).

▷ For more information on this or any other computing topic, please visit our Digital Library at [www.computer.org/csdl](http://www.computer.org/csdl).