RAT: A Refactoring-Aware Traceability Model for Bug Localization

Feifei Niu*, Wesley K. G. Assunção†, LiGuo Huang‡, Christoph Mayr-Dorn†,
Jidong Ge*, Bin Luo*, Alexander Egyed†
*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China
Email: niufeifei@smail.nju.edu.cn, {gjd, luobin}@nju.edu.cn
†Institute for Software Systems Engineering, Johannes Kepler University, Linz, Austria
Email: {wesley.assuncao, christoph.mayrdorn, alexander.egyed}@jku.at
‡Department of Computer Science and Engineering, Southern Methodist University, Dallas, Texas, USA
Email: lghuang@lyle.smu.edu

***Abstract*—A large number of bug reports are created during the evolution of a software system. Locating the source code files that need to be changed in order to fix these bugs is a challenging task. Information retrieval-based bug localization techniques do so by correlating bug reports with historical information about the source code (e.g., previously resolved bug reports, commit logs). These techniques have shown to be efficient and easy to use. However, one flaw that is nearly omnipresent in all these techniques is that they ignore code refactorings. Code refactorings are common during software system evolution, but from the perspective of typical version control systems, they break the code history. For example, a class when renamed then appears as two separate classes with separate histories. Obviously, this is a problem that affects any technique that leverages code history. This paper proposes a refactoring-aware traceability model to keep track of the code evolution history. With this model, we reconstruct the code history by analyzing the impact of code refactorings to correctly stitch together what would otherwise be a fragmented history. To demonstrate that a refactoring aware history is indeed beneficial, we investigated three widely adopted bug localization techniques that make use of code history, which are important components in existing approaches. Our evaluation on 11 open source projects shows that taking code refactorings into account significantly improves the results of these bug localization techniques without significant changes to the techniques themselves. The more refactorings are used in a project, the stronger the benefit we observed. Based on our findings, we believe that much of the state-of-the-art leveraging code history should benefit from our work.**

***Index Terms*—bug localization, bug report similarity, code refactoring, traceability, commit history, information retrieval**

I. Introduction

Software maintenance runs through the entire software lifecyle [1] [2], among which, debugging and fixing bugs is the dominant task [3]. In response, it has become popular for projects to use bug tracking systems (e.g., Bugzilla, JIRA) to collect and manage bug reports. However, bugs happen in bursts and a sizable number of bug reports are often filed on any given day. For example, Anvik et al. report that Eclipse received up to 192 bug reports the day the project was released [4] and then a steady stream of reports thereafter. Consequently, one of the most time-consuming tasks is in understanding the bugs and where in the source code they might need to be fixed. Over 15 years of studies reveal that 40% of the cost of software maintenance is trying to understand the software for bug localization [5].

To cope with this problem, a vast amount of research has been done to automatically locate buggy code files. Information retrieval-based (IR-based) bug localization is a widely employed technique that takes bug reports and source code as input, and outputs a ranked list of files that may cause the bug. State-of-the-art approaches leverage the code evolution history to improve on accuracy, for example: similar reports [6]–[10], commit history [7]–[15], source code [6], [7], [10], [13], [16], [17], and stack trace [10], [18]. Indeed, similar reports and commit history are two of the most commonly used input for these approaches. For example, both Zhou et al. [6] and Rath et al. [8] found that bug reports that are similar typically change the same files. This led to a stream of research on creating similarity reports (bug reports and enhancement reports/feature requests). The idea is simple. If a new bug is similar to a previously solved bug, then the previously changed/fixed code files can be used as a basis for predicting the files that need changing now. These techniques assign a suspicious score to each file that has been changed by resolved bug reports based on textual similarity. If a file has been fixed more than once, then the scores are aggregated. This requires understanding the history of the files. Alternative techniques have been created that more directly leverage code history. For example, commit history-based approaches found that most frequently changed files or files that changed most recently are more likely buggy and require fixing in the near future [14]. It collects the commit history of each source file and then calculates a suspicious score for that file. There are other techniques that also leverage code history for bug localization, but these two kinds of approaches presented above have been proven to be effective and have been widely adopted.

One key problem that all existing approaches appear to share is that they use absolute path or fully-qualified class names to uniquely identify source code files. For any given point in time of the code history, this is a safe thing to do because a fully qualified name makes a code file unique if there are multiple files with the same name (not uncommon). However, when observing a file over the course of its history, doing

so becomes problematic. If a file is renamed, then the fully-qualified name changes. Likewise, if a file is moved from one package to another. From the perspective of the version control systems, such changes are equivalent to file deletions and file creations. That is, a file that is moved from one package to another will appear to have been deleted from the one package and a different file will appear to have been created in the other package. As a result, the history of that moved file will be broken, as it ends in the file that now appears as deleted and restarts in the file that now appears as created. Since similarity bug report-based and commit history-based approaches rely on the source file history, they are naturally affected by this. A file will appear to be less frequently changed or a file will appear to be related to fewer similar bug reports. Even co-relations to bug reports are affected. If a code file was changed to fix a bug and thereafter the file was moved from one package to another, then the bug report remains linked to the old, now apparently deleted file. Hence, this link becomes obsolete and existing bug localization techniques ignore obsolete links for obvious reasons. No matter the situation, a code file with a broken history will suffer in any bug localization technique because its history will appear shorter or its relationship will appear to be outdated.

The main culprit that breaks code history is code refactoring. Code refactoring is a well known, studied, and even encouraged activity [19]. Code refactoring restructures the source code without changing the behavior. Developers refactor their source code to mitigate code smell [20], for better readability [21], ease of maintenance and evolution, etc. However, code refactoring breaks traceability of code between different versions. For instance, "Move Package" and "Rename Class" will change the file path. "Move and Rename Method" will break method level history. By far, existing studies on code refactoring mainly focus on detecting code refactoring from source code [22]–[27], analyzing relationship between code refactoring and bugs (e.g., whether refactoring introduces bugs [28]–[30], whether refactoring improves code readability [22], [31] or removes code smell [19], [20], [30], [32]). However, understanding of code history with the goal of improve code traceability or even bug localization is still an explored topic.

In this paper, we propose a code traceability model, to store the history of code elements (code blocks) not just in terms of time revisions but also in terms of refactorings. We employ refactoring mining tools to detect code refactorings and then reconstruct the code history. A file that has been moved from one package to another will continue to be one file in our history where the move is now one part of the file's history, with all the changes that happened before and after appearing in the file's history. This paper then investigates whether a more complete/less fragmented code history can improve the accuracy of bug localization. To evaluate this, we select three widely-adopted bug localization approaches as our baselines, i.e., SimiScore [6], TraceScore [8], BugCache [15], [33], which are usually adopted by existing approaches as one of their components. We performed extensive evaluations to answer the following research questions:

- **RQ1**. Do code refactorings benefit bug localization?
  -1.1. Do code refactorings benefit bug localization based on bug report similarity?
  -1.2. Do code refactorings benefit bug localization based on recently more frequently modified files?
- **RQ2**. How often do code refactorings happen?

Our experimental results based on 11 open source projects with more than 8,000 bug reports show that existing approaches greatly benefit from integrating our refactoring-aware code history. When taking refactoring into consideration, the MAP of SimiScore, TraceScore (one-year history) and TraceScore-W (whole history) is improved by 28.5%, 9%, and 20%, respectively. For BugCache, the MAP of most projects is improved by 20% to 37%. In general, the more the project was refactored, the more significant are the improvements.

The main contributions of this work are as follows.

- We propose a Code Refactoring-Aware Traceability Model, named RAT, based on a data model to reconstruct code evolution history that may have been broken by code refactoring. To our best knowledge, we are the first to employ code refactoring to reconstruct code evolution history and recover code traceability.
- We performed empirical evaluations on three widely-used bug localization approaches to demonstrate the improvement of bug localization performance by adopting our model over the baselines.

The rest of this paper is organized as follows. Section II introduces the background upon which our approach is built. Section III motivates our work. Section IV elaborates our approach. Section V evaluates the proposed approach. Section VII discusses threats to validity. Related work is presented in Section VIII. Section IX concludes the work.

II. BACKGROUND

In this section, we first explain the necessary background for better understanding IR-based bug/fault localization approaches and techniques that can potentially be affected by code rename refactoring. Then, we present background about code refactoring that breaks the file history.

A. Bug Localization Approaches

IR-based bug localization methods leverage bug reports and source codes as input and search for suspicious files/methods that may be buggy. The input of mainstream approaches can be classified into three types: 1) calculate the textual similarity between source code and bug report texts [6], [7], [10], [13], [16], [17], 2) utilize similar reports that have been fixed [6]–[10], and 3) make use of commit history which maintains a relatively short list of the recent most fault-prone files for prediction [7]–[15]. There can also be other inputs like stack trace [10], [18], [34]. Then the mainstream approaches assign weights to different suspicious scores to output the final list [7], [8], [10], [16], [35], [36]. Among these, bug report similarity and version history are widely-adopted by mainstream approaches as one of their components, and are potentially most influenced by code history. Table I shows the

weights assigned to bug report similarity and version history in different approaches, except for AmaLgam+ [10] and ABLoTS [8], which utilize genetic algorithm [37] and decision tree to learn weights.

TABLE I
WEIGHTS IN EXISTING APPROACHES

Approaches	Bug Report Similarity (SimiScore/TraceScore)	Version History (BugCache)
BugLocator [6]	0.2-0.3	0
BLUiR+ [16]	0.2	0
AmaLgam [7]	0.14	0.3
BRTracer [18]	0.2	0
BLIA [9]	0.12-0.32	0.2-0.4
AmaLgam+ [10]	-	-
BLIA+ [34]	0.058-0.065	0.07-0.17
ABLoTS [8]	-	-

Bug Report Similarity-based Techniques. Similar bug reports are more likely to be linked to the same files [6]. For a new bug report, this technique calculates textual similarity to find similar bug reports or requirements that have been fixed or implemented. Then based on these, it recommends suspicious files. There are a lot of approaches to adopt and benefit from similar bug reports [6], [8].

SimiScore [6] and TraceScore [8] are two of the widely-adopted and state-of-the-art similar-reports-based techniques. Both treat the summary and description of a new bug report $b*$ as query, summary and description of previously resolved bug reports as documents, and search for similar artifacts A ($b* \notin A$). Files that have been modified to fix or implement A are S_A. Based on the textual similarity of $b*$ and each $a \in A$, the file $s \in S_A$ inherits a suspicious score from the textual similarity. Then the suspicious scores of the candidate files are ranked in an incremental list of bug-prone files. There is a slight difference between SimiScore and TraceScore. SimiScore leverages all the previously resolved bug reports and calculates the score according to (1), where $sim(a_i, b*)$ represents the textual similarity between a_i and b*. TraceScore involves all types of issues (including bug reports and requirements), but only traces back bug reports filed within one year and filters out big issues that modify more than 10 files. It calculates the score of files by (2).

$$SimiScore(s, b*) = \sum_{a_i \in \{a | s \in fix(a)\}} \frac{sim(a_i, b*)}{|fix(a_i)|} \quad (1)$$

$$TraceScore(s, b*) = \sum_{a_i \in \{a | s \in fix(a)\}} \frac{sim(a_i, b*)^2}{|fix(a_i)|} \quad (2)$$

Recent More Frequent Modification favored Techniques. Kim et al. hypothesized that past knowledge of bug occurrence can optimize bug inspection [14]. They use a "cache" to maintain a list of most bug-prone files. Rahman at al. [12] found out that the bug density of files in the "cache" is higher than files outside the "cache". They propose an algorithm which sorts files based on the bug fixing commits. Google's developers adopt this algorithm to predict bugs on their projects [15] [33], which have been widely adopted and well tested [7], [8], [10], [34]. Google's BugCache maintains the commit history of each file in the system by the day of a new bug fix. BugCache identifies bug-fixing commits C based on the appearance of 'bug' and 'fix' in the commit message. Then they calculate the cache score based on (II-A), where f is one of the buggy files in commit $c \in C$, t_i is the elapsed time in days since the file's creation. This formula tends to select the most recently and frequently changed files.

$$BugCache(f) = \sum_{c \in C \wedge f \in c} \frac{1}{1 + e^{-12(t_i)+12}} \quad (3)$$

Since the above two techniques use the past history, they are more likely to be influenced by the code history. For example, a change in a file f fixing a bug b results in a link between f and b. If file f was then renamed to f', f will no longer exist in the source code. But b is still linked to the now non-existent f, making this link obsolete. However, if we are aware that f' is f, then we can relink b to f', to update the information.

B. Code Refactoring

Code refactoring was first introduced by Martin Fowler as the activity of improving the internal quality of source code without changing the functional behavior of the software [19]. There are more than 80 types of different refactoring, concerning moving, renaming and extracting packages, classes, methods, attributes and so on. Refactoring has been thought to be useful for relieving code smells [20], [30], [32], and also notably affects code readability [21]. Thus, code refactoring has been continuously adopted by developers.

A robust body of research has emerged to better understand the influence of refactoring. Di Penta et al. [28] and Bavota et al. [29] reveal the high odd to induce bug-fixing for refactoring types involve inheritance and refactoring big chunks of code. Halepmollasi and Tosun also discover commits that perform refactoring are three times more fault-inducing than other commits [30]. Brito et al. proposed the concept of refactoring graphs to improve code comprehension and support software evolution studies [22] [31]. They use a graph to visualize the transformations performed by refactoring over time. Refactoring graphs focus on method-level refactoring, not all the code elements, and on short-term refactoring, not the code history.

In terms of detecting code refactoring, some useful tools have been proposed, including RefDiff [24], [25] and RefactoringMiner [27] [26]. As far as we are concerned, RefactoringMiner 2.0 [26] is the state-of-the-art, can detect more than 80 types of code refactoring, has the highest average precision of 99.7% and recall of 94.2%, as well as being faster than other tools. The implementation is publicly available on GitHub.[1] In this paper, we used RefactoringMiner 2.0 as our tool to identify code refactoring types. We include only the eight types that directly affect the files' history: "Move Class", "Rename Class", "Move and Rename Class", "Rename Package", "Move Package", "Split Package", "Merge Package" and

[1] https://github.com/tsantalis/RefactoringMiner

"Move Source Folder" (RefactoringMiner 2.0 is able to detect source folder movement). Definitions of these refactorings can be found in Fowler's book [19].

III. Motivating Example

In this section, we illustrate an example that motivates our approach. Fig. 1 shows an open bug report HORNETQ-578 from project Hornetq[2] and three resolved bug reports (i.e., HORNETQ-311, HORNETQ-812, HORNETQ-911) that have the highest similarity to HORNETQ-578 in the resolved bug reports. Their similarity scores are 0.2703, 0.2194, 0.2330, respectively. Note that the textual similarity scores are calculated based on both summary and description in the bug reports. In Fig. 1, we only show the ID and Summary of each bug report, omitting the description due to the space limit. For resolved bug reports, we also present the Created Date, Resolved Date, and Modified Files to fix the bugs.

In Fig. 1, we also list the eight commits that occurred during this period of time, and six of them were committed to resolve the three bug reports. The three connected horizontal timelines aligning three commit sequences represent the evolution history of the three different but related files that were changed by these commits. During code evolution, the full path is commonly included in the filename to uniquely identify a file. Initially, the file "**src/main/**org/.../JMSServerManagerImpl.java" was created. Then, after commit c_2, it was renamed to "**hornetq-jms/src/main/java/**org/.../JMSServerManagerImpl.java" by "Moving Source Folder" operation. Afterwards, at commit c_8, it was renamed again by another "Moving Source Folder" operation to "**hornetq-jms-server**/src/main/java/org/.../JMSServerManagerImpl.java". We observed that the three parallel timelines in Figure 1 were linking the commits to the same file, with different names (and full paths) at different times in the evolution history.

Let's use SimiScore [6], a bug locator, to illustrate how existing bug localization approaches use text similarity to locate relevant files to fix the bug in the open bug report HORNETQ-578. SimiScore believes that fixing bugs in similar bug reports tends to change similar files [6]. In our example, seven files were changed by the three resolved bug reports and are the candidate files evaluated by SimiScore for the most likely changed file to fix HORNETQ-578. Since fixing HORNETQ-311 changed two files, the SimiScore for each file was calculated as $simiscore(b*, f_1) = simiscore(b*, f_2) = 0.2703/2 = 0.135$. Similarly, we can calculate the similarity score for each of the seven candidates. The file "hornetq-jms/src/main/java/org/.../JMSServerManagerImpl.java" (renamed after commit c_2) was one of the 4 files changed to fix HORNETQ-812 and one of the 2 files changed to fix HORNETQ-911. Its SimiScore is $0.2194/4 + 0.2330/2 = 0.171$, the highest similarity score among the seven candidates. As a result, this file was recommended by SimiScore for change to fix the open bug. It was indeed the file that was most likely to contain the bug described in the open bug report. However, the recommended file with the old filename no longer exists after being renamed again by commit c_8. The issue was caused by the fact that SimiScore treated the file being renamed twice at commits c_2 and c_8 as three different files, the source file tracking history being broken by refactoring operations that renamed files. One may argue that removing renamed files with old filenames would solve the problem. However, newly renamed files are often not changed as much as files with longer history, so their SimiScores are often lower than older files. Our example echoed this. Even if we only included the most recently renamed file "**hornetq-jms-server**/src/main/java/org/.../JMSServerManagerImpl.java" into the SimiScore comparison, other older files with higher SimiScore would be falsely recommended.

In this case, if a bug localization approach such as SimiScore leverages code evolution history unaware of the code refactoring, such as file renaming, it will consider the renamed files as different ones and separate the similarity analysis for each of them. Therefore, it may make inaccurate recommendations of the file most likely to be changed for an open bug. If the approach could take file renaming into consideration, it would be able to reconstruct and keep track of the complete file evolution history and make a more accurate recommendation.

With this example, we demonstrate the importance of considering file renaming into file-level bug localization. We show that "moving source folder" would change the file path, thus breaking the tracking history of the file, but there are many types of operations in the development process that could similarly cause bias to the bug localization approaches that utilize the code evolution history. Other types of rename refactoring, including move package, rename package, rename class etc., may have an equivalent impact. If we could reconstruct the history of the source file and recover the tracing links between renamed files, we would be able to get rid of such bias.

IV. Approach

In this paper, we propose a code refactoring-aware traceability model, which is open source and free available [3], to capture the code change history. We use code refactoring detection tools to recover code history that may be broken by code refactoring. With this model, we can reconstruct the whole history of source code, which is called Code Block History. Then, we apply Code Block History in both bug report similarity-based and commit history-based bug localization. The framework is as shown in Fig. 2. Firstly, we would illustrate the concept of the code traceability model. Then we will demonstrate how to construct code block history from source code. Finally, we will show how to leverage code block history in bug localization.

A. Code Refactoring-Aware Traceability Model

Fig. 3 presents the class diagram of the model. There are different code snippets in a source code, i.e., package, class,

[2]https://github.com/hornetq/hornetq

[3]https://github.com/feifeiniu-se/traceability

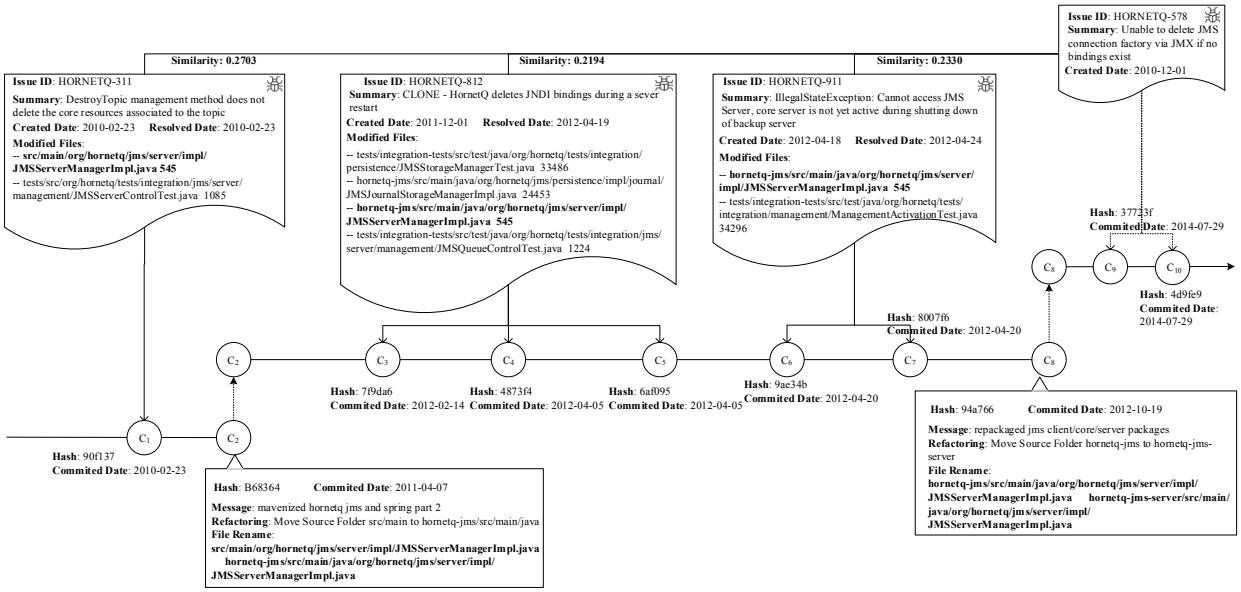

Fig. 1. Motivating Example

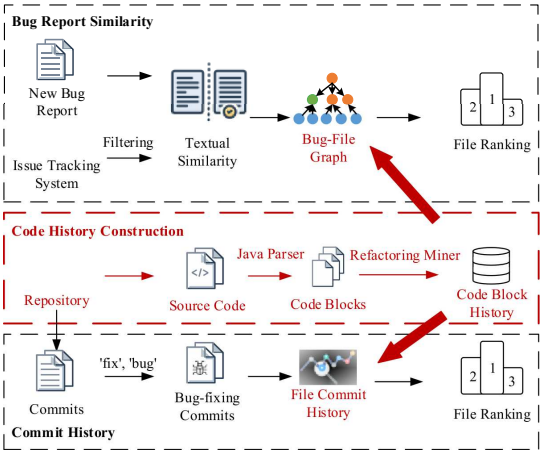

Fig. 2. Framework for Leveraging Code Refactoring-Aware Traceability Model into Bug Localization.

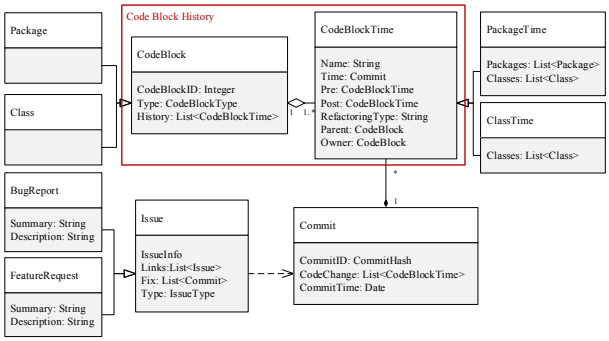

Fig. 3. Code Refactoring-Aware Traceability Model

method, attribute, parameter and variable, which we call code block. In this paper, we mainly focus on packages and classes according to the localization granularity. Every project is made up of many code blocks. To identify them, we assign them the unique ID **CodeBlockID**. One code block can be a package or a class, denoted by **Type**. During the evolution of software, there can be thousands of commits, each potentially renaming, moving some code blocks to a different package, etc. Thus, the same code block may have different names at different times, **yet, the file path is by far the most common identification in existing bug localization approaches.** To capture the different external appearances of code blocks at different commit times, we use **CodeBlock.History** to store the whole life cycle of code blocks, which consists of a series of **CodeBlockTime**. Each $CodeBlockTime$ stores information of code block at commit c, which includes name, time, refactoring type, parent code block and child code blocks etc. $PackageTime$ and $ClassTime$ are inherited from $CodeBlockTime$, indicating history of different types of code blocks. $CodeBlockTime$ ensures to keep track of the same source code throughout name-change refactorings. To facilitate collecting code information at commit c, we also use **Commit**, which consists of list of $CodeBlockTime$ at commit c. We use the commit hash to indicate the commit ID.

Given a $CodeBlockID$, we can easily go through the history with the Pre and $Post$ link in $CodeBlockTime$. One block of code can be renamed many times in the history, so the $name$ attribute records the name changes at different commits. Given a commit hash, we can collect all the code blocks that exist at that time. What's more, $CodeBlockTime$ and $Commit$ are linked by the $CodeBlockTime.Time$ and $Commit.CodeChange$. Externally, commits are linked to issues to fix bugs or implement requirements. All these trace links make up the traceability model, with which we can easily keep both horizontal and vertical traceability of code change.

B. Collecting Code Block History

With the above-mentioned traceability model, we can keep track of the code history, called Code Block History. The main idea of code block history is to obtain the code blocks from the source code. Then for each source code, capture its information throughout the history. The key point here is to link a new $CodeBlockTime$ to the previous $CodeBlockTime$. For most of the $CodeBlockTime$, the block name is indicative. We can link different $CodeBlockTime$s based on the same name. However, code refactoring would change the name of the code block during the commit. It would be tricky if we link different versions of $CodeBlockTime$ merely according to the name. A lot of tools have been proposed to detect the code refactorings [24], [24], [26], [27]. Thus we adopt a refactoring miner to deal with our tricky problem.

Specifically, as shown in the red box in Fig. 2, given one version of source code files S_t at time t, firstly, we use JavaParser[4] to parse all the source files, and obtain all the code blocks, i.e., packages and classes. For each code block, we would create a $CodeBlock$ to store all the information of the block, and the first $CodeBlockTime$ to store the information at time t. There can also be some files that can not be parsed, or they do not have java class, we would also create $CodeBlock$ for such files, to make sure that we do not miss files. The name of such code blocks is set to be the file path.

Then, for the two versions of source code S_t, S_{t+1} before and after one commit C, we parse the source code, and obtain the collection of code block time in the two versions of source code, which we denote by CBT_t and CBT_{t+1}. For each $cbt_i \in CBT_t$, $cbt_j \in CBT_{t+1}$, if $cbt_i.name == cbt_j.name$, then $cbt_i.post = cbt_j$, $cbt_j.pre = cbt_i$. Then we use the state-of-the-art refactoring miner tool [26] to mine the refactorings between S_t and S_{t+1}. With the detected refactorings, we would be able of know pairs of cbt_m, cbt_n ($cbt_m \in CBT_t$, $cbt_n \in CBT_{t+1}$, $cbt_m.name \neq cbt_n.name$), that satisfy cbt_m is renamed to cbt_n. Thus, we can have the links: $cbt_m.post = cbt_n$, $cbt_n.pre = cbt.m$. So far, we are able to link different $CodeBlockTime$s with refactorings, even if they are renamed. The links between commits and issues are already connected to each other by the open-sourced bug localization dataset.

C. Leveraging Code Refactoring-Aware Traceability Model in Bug Localization

Fig. 2 shows the framework for employing a code refactoring-aware traceability model in bug localization techniques. Our aim of this paper is to investigate whether existing bug localization techniques can benefit from using code refactorings. We select two different kinds of well-known techniques: *bug report similarity* based and *recent more frequent change favored* techniques. For bug report similarity-based techniques, we use the widely used SimiScore [6] and state-of-the-art TraceScore [8] as examples. For recent more frequent change favored techniques, we choose the well-tested google's implementation of BugCache [15] [33]. These three approaches have been successively applied in dozens of researches [6]–[10], [18]. Details of these three techniques are explained in Section II-A. For a fair comparison, we do not change the approaches and use the same configuration and parameters. We only translate the file path into Code Block ID. In this way, even though some refactored files may have different paths, they still share the same Code Block ID.

For a given file path, we have to recognize the package and the class name, then search for the corresponding $CodeBlockTime$, and then get the Code Block ID. In Fig 2, the "Bug-File Graph" part would be constructing the graph with ID not the file path. The "File Commit History" part would be calculating the BugCache of $CodeBlock$s, not file path. In theory, the mapping does not change the approaches, but only merges the score of the renamed files.

We sort all the bug reports by the resolved time. For a new bug report, we select resolved issues and bug-fixing commits as historical information. For SimiScore, it only applies previous bug reports, while TraceScore utilize both bug reports and feature requests. BugCache uses bug-fixing commits, which are filtered by the keywords "fix" or "bug" in the commit message.

V. Experiments

To investigate how code refactorings would influence components and approaches that involve code history, we answer two research questions, which assess the influence of employing code refactorings. Then we evaluate our approach from different practical perspectives.

A. Data Set

Rath et al. [8] collected a dataset consisting of 15 open-sourced projects. We reused 11 out of them to evaluate our approach. In the other 4 projects, some commits have been forked by the developer and no longer available in the repository, while BugCache requires the commit history. For fair comparisons, we exclude these four projects from the dataset. In this dataset, there are issues (including bug reports and feature requests), commits that are committed to resolve the issues, and files that are modified by the commits, making up the trace links in the lower part of Fig 3.

Apart from the original information provided in the dataset, we also collect all the **commit logs** from Git repositories of those 11 projects, which is the input of BugCache.

Lastly, and most importantly, we collect **Code Block History** during the studied time period, as described in Section IV-A. Firstly, we download git repositories of the 11 projects from GitHub. Then we detect code refactorings with RefactoringMiner 2.0 tool[5] proposed by Nikolaos et al. [26]. RefactoringMiner 2.0 is able to detect more than 80 types of refactoring, but we only select 8 types that would rename files. With the output, we would be able to know the traceability

[4]https://github.com/javaparser/javaparser

[5]https://github.com/tsantalis/RefactoringMiner

between renamed files or classes. We parse all the source code files at the initial commit, create code blocks and construct the $CodeBlock.History$ for each code block.

Finally, there are 8,494 bug reports and 6,029 requirements in the dataset. Table II describes the dataset in more detail.

B. Experiment Design

In order to investigate whether bug localization can benefit from code refactoring (RQ1), we select different types of bug localization approaches, that are based on different information, i.e., bug report similarity(RQ1.1) and recent more frequent modified files(RQ1.2).

1) Bug Report Similarity: Bug report similarity-based approaches retrieve resolved issues (i.e., bug reports and feature requests) from the history. We select SimiScore proposed by Zhou et al. [6], TraceScore by Rath et al. [8], as our baseline, with the same configuration proposed by the authors.

SimiScore calculates the textual similarity between the new bug report and resolved bug reports from the whole history. TraceScore uses both resolved bug reports and feature requests, but it only uses bug reports and feature requests that are fixed within the last year. They filter out big issues that change more than 10 files. To see how the length of history would be influenced, we add the third baseline that tests on all the resolved bug reports and feature requests for TraceScore. More details about calculation are described in Section II-A.

For comparison, we add code history in the three baseline experiments. When taking code history into consideration, the original links from bug reports to modified files, would become bug reports to $CodeBlockIDs$. Then we evaluate the three groups of experiments with the evaluation metrics.

2) Recently More Frequently Modified File Favored: This kind of approach utilizes commit history to find buggy-prone files, which is proposed and verified by [14] [12]. Commits also come from the past, and thus may be influenced by code refactoring. We select the well-tested and commonly-used Google's implementation of BugCache [15] [33] as the baseline. We collect all commit logs from the code repository. Then pick out bug-fixing commits by the appearance of keywords "fix" and "bug" in the commit message. Then for all the files in each commit, we use the formula (II-A) in Section II-A to calculate the suspicious score.

It is worth noting that for BugCache, instead of carrying out comparison experiments on all the bug reports in the dataset, we test on bug reports who modify files that have been renamed during the history. In this way, we would like to see how our code history model works when dealing with refactored files.

C. Evaluation Metrics

To evaluate the effectiveness of the approach, we adopt commonly used metrics as follows:

Top@k (Hit@k) measures the percentage of bug reports in which at least one of the buggy files is a top *k* ranked file.

Mean Average Precision (MAP) is a standard metric widely used in information retrieval to evaluate ranking approaches. It considers all the ranks of all buggy files into consideration. It is calculated as the mean of the Average Precision over all queries. Average Precision of a given bug report aggregates precision of positively recommended files as:

$$AP = \sum_{i=1}^{N} \frac{P(i) * pos(i)}{\#\,of\,positive\,instances} \tag{4}$$

where i is a rank of the ranked files, N is the number of ranked files and pos(i) $\in \{0,1\}$ indicates whether the *i*th file is a buggy file or not. P(i) is the precision at a given top i files:

$$P(i) = \frac{\#\,of\,buggy\,files\,in\,top\,i}{i} \tag{5}$$

Mean Reciprocal Rank (MRR) computes the average of the reciprocal of the positions of the first correctly located buggy files in the ranked files, following this equation:

$$MRR = \frac{1}{Q} \sum_{|Q|}^{q=1} \frac{1}{rank_i} \tag{6}$$

VI. Experiment Results

RQ1 investigates whether our approach can improve the accuracy of bug localization. Then we verify if rename refactorings are a common practice in the subject systems of our study at RQ2. Fig. 4 presents the number of files of each system and highlights the percentage of files that underwent rename refactorings (at the top of the columns). We have a variety of results, from projects with few refactorings (e.g., Railo with only 1%) to projects with intensive application of refactorings (e.g., Hornetq with 63%). This diversity of values allows us to see how bug localization benefits from refactorings.

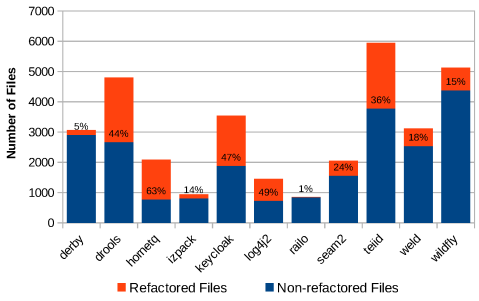

Fig. 4. Proportion of files with rename refactorings.

A. RQ1.1. Do code rename refactorings benefit bug localization based on bug report similarity?

The results to answer this RQ are shown in Table III. For each project, we can observe the difference in the bug localization techniques based on bug report similarity. From the table, we can see that considering rename refactorings (i.e., +R) leads to improvements to all three baselines, or at least equal results. There were only two cases in each comparison with equal results, both for the system Railo, which is the system with the least proportion of rename refactorings (i.e.,

TABLE II
CHARACTERISTICS OF THE DATASET

Project	#Bug Reports	Changed Source Code Files Per Bug Rport				#Requirements	Changed Source Code Files Per Requirement				#Commits	#Files	#Code Blocks	#Dependency Trace Links	#Bug Reports Touch Renamed Files
		min	median	mean	max		min	median	mean	max					
Derby	1778	1	2	6.5	1920	1264	1	4	14.4	1334	5255	3224	3065	1764	433
Drools	1281	1	3	17.4	1371	653	1	11	76.7	3247	3682	6933	4799	153	1211
Hornetq	270	1	4	7.4	50	187	1	10	59.5	3680	1003	3395	2087	42	265
Izpack	318	1	3	9	140	160	1	8	24	460	927	1077	943	49	199
Keycloak	786	1	4	11.3	645	637	1	10	39.8	4652	2753	5196	3540	360	621
Log4j2	441	1	2	7.6	385	335	1	4	31.7	1266	1574	2173	1454	200	399
Railo	300	1	2	4	66	129	1	4	8.5	140	470	871	859	7	34
Seam2	776	1	1	2.7	63	526	1	3	12.8	2268	1477	2539	2049	246	721
Teiid	1297	1	3	14	1073	1162	1	8	72.9	3616	3462	8118	5949	311	1003
Weld	560	1	4	8.7	570	419	1	7	25	2050	1460	3696	3119	228	388
Wildfly	687	1	2	7.7	295	557	1	8	37.9	3270	2266	5878	5130	1925	398

1%) (see Fig. 4). This indicates that all three baselines do benefit more or less from using more complete code history constructed by RAT, on our dataset.

Fig. 5 presents the values of the metrics MAP (at the top) and MRR (at the bottom) for each project. In this figure, we compare the results of three baselines (i.e., SimiScore, TraceScore, and TraceScore with whole history) without and with considering refactoring information (+R). Since we aim to understand the influence of refactorings, in this figure we sorted the projects according to the renaming rate.

From the figure, we can observe an increase with using refactorings in all three baselines for MAP and MRR (except for MAP of Railo, which remains the same). What is also obvious is that from left to right (renaming rate of projects increases), the improvement in MAP and MRR value becomes more significant. Project Railo (1% renaming rate) has almost no improvement, while Hornetq (63% renaming rate) has the most notable improvement. This indicates that, when project has higher rename rate, more complete code history becomes more helpful for better accuracy.

To corroborate the analysis, we computed the Pearson correlation coefficient. Table IV presents the correlation and p-values comparing the results of MPA and MRR and the renaming rate. In all cases, there is a strong positive correlation, which indicates that the more renaming refactorings in the projects, the more significant improvement in MAP and MRR results. Since all p-values are lower than 0.001, we have 99% of confidence in these results.

When comparing without and with considering refactorings, for SimiScore, the improvement varies from 2% to 134% for MAP and from 2% to 100% for MRR on all projects. Average MAP and MRR are improved by 28% and 20%. For TraceScore, the improvement varies from 0% to 21% for MAP and 0% to 16% for MRR. Average MAP and MRR are improved by 9% and 8% on all projects. For TraceScore with whole history (TraceScore+W), the improvement varies from 0% to 105% for MAP and 1% to 91% for MRR. Average MAP and MRR are improved by 20% and 19%. When compared with the other two baselines, TraceScore (with one-year history) shows less improvement. The reason is that

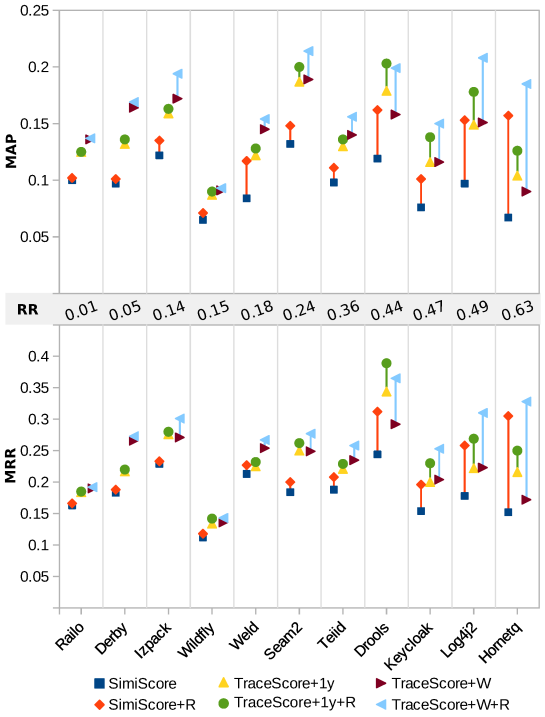

Fig. 5. MAP and MRR for each project. The projects appear sorted by the Refactoring Rate (RR)

TraceScore only involves one-year history, while SimiScore and TraceScore+W involve whole history (which is around 10 years). With shorter history, file renaming happens less and therefore less improvement. This shows that our RAT model is particularly effective for approaches that need to exploit long histories. TraceScore is an advanced version of SimiScore by Rath et al. [8], which also takes user requirements and trace links into consideration, thus we can observe higher accuracy in TraceScore. With longer history, TraceScore+W can leverage more information, thus gains improvement than TraceScore in most of the projects. However, in some projects, for example, Drools, we observe a degradation in accuracy, which is consistent with the reported result in the original pa-

TABLE III
PERFORMANCE OF TECHNIQUES USING BUG REPORT SIMILARITY WITH AND WITHOUT CONSIDERING RENAME REFACTORINGS.

Project	Approach	MAP	MRR	Top 1	Top 5	Top 10
Derby	SimiScore	0.097	0.183	0.106	0.260	0.335
	SimiScore+R	0.101	0.188	0.110	0.262	0.342
	TraceScore	0.132	0.217	0.140	0.294	0.367
	TraceScore+R	0.136	0.220	0.142	0.301	0.372
	TraceScore+W	0.164	0.266	0.169	0.364	0.458
	TraceScore+W+R	**0.169**	**0.273**	**0.175**	**0.374**	**0.470**
Drools	SimiScore	0.119	0.244	0.137	0.351	0.458
	SimiScore+R	0.162	0.312	0.171	0.473	0.572
	TraceScore	0.179	0.344	0.233	0.479	0.568
	TraceScore+R	**0.203**	**0.389**	**0.273**	**0.529**	0.608
	TraceScore+W	0.158	0.292	0.186	0.406	0.507
	TraceScore+W+R	0.199	0.365	0.226	0.521	**0.628**
Hornetq	SimiScore	0.067	0.152	0.081	0.222	0.322
	SimiScore+R	0.157	0.305	0.213	0.418	0.515
	TraceScore	0.104	0.216	0.131	0.291	0.407
	TraceScore+R	0.126	0.250	0.157	0.332	0.451
	TraceScore+W	0.090	0.172	0.090	0.250	0.343
	TraceScore+W+R	**0.185**	**0.328**	**0.224**	**0.437**	**0.541**
Izpack	SimiScore	0.122	0.229	0.151	0.308	0.390
	SimiScore+R	0.135	0.233	0.129	0.331	0.438
	TraceScore	0.159	0.276	0.186	0.382	0.451
	TraceScore+R	0.163	0.280	**0.189**	0.388	0.473
	TraceScore+W	0.172	0.271	0.174	0.379	0.489
	TraceScore+W+R	**0.194**	**0.301**	0.180	**0.432**	**0.565**
Keycloak	SimiScore	0.076	0.154	0.084	0.220	0.302
	SimiScore+R	0.101	0.196	0.115	0.267	0.372
	TraceScore	0.116	0.200	0.119	0.287	0.372
	TraceScore+R	0.138	0.230	0.136	0.336	0.424
	TraceScore+W	0.116	0.204	0.118	0.290	0.377
	TraceScore+W+R	**0.150**	**0.253**	**0.153**	**0.364**	**0.454**
Log4j2	SimiScore	0.097	0.178	0.113	0.249	0.308
	SimiScore+R	0.153	0.258	0.169	0.347	0.441
	TraceScore	0.149	0.222	0.148	0.295	0.390
	TraceScore+R	0.178	0.269	0.183	0.370	0.468
	TraceScore+W	0.151	0.223	0.135	0.308	0.388
	TraceScore+W+R	**0.208**	**0.310**	**0.199**	**0.436**	**0.527**
Railo	SimiScore	0.100	0.163	0.100	0.217	0.287
	SimiScore+R	0.102	0.166	0.101	0.221	0.295
	TraceScore	0.125	0.184	0.114	0.248	0.339
	TraceScore+R	0.125	0.185	**0.114**	0.252	0.342
	TraceScore+W	0.136	0.190	0.111	0.282	0.356
	TraceScore+W+R	**0.137**	**0.192**	0.111	**0.292**	**0.356**
Seam2	SimiScore	0.132	0.184	0.117	0.245	0.316
	SimiScore+R	0.148	0.200	0.125	0.271	0.357
	TraceScore	0.187	0.250	0.161	0.351	0.427
	TraceScore+R	0.200	0.262	0.173	0.355	0.446
	TraceScore+W	0.189	0.249	0.165	0.339	0.424
	TraceScore+W+R	**0.214**	**0.277**	**0.182**	**0.377**	**0.476**
Teiid	SimiScore	0.098	0.188	0.109	0.252	0.357
	SimiScore+R	0.111	0.208	0.125	0.278	0.383
	TraceScore	0.130	0.221	0.127	0.320	0.414
	TraceScore+R	0.136	0.229	0.132	0.329	0.425
	TraceScore+W	0.140	0.235	0.137	0.331	0.446
	TraceScore+W+R	**0.156**	**0.258**	**0.159**	**0.361**	**0.478**
Weld	SimiScore	0.084	0.213	0.139	0.273	0.364
	SimiScore+R	0.117	0.227	0.144	0.300	0.389
	TraceScore	0.122	0.225	0.138	0.322	0.407
	TraceScore+R	0.128	0.232	0.142	0.326	0.424
	TraceScore+W	0.145	0.254	0.158	0.350	0.455
	TraceScore+W+R	**0.154**	**0.267**	**0.169**	**0.361**	**0.484**
Wildfly	SimiScore	0.065	0.112	0.071	0.148	0.188
	SimiScore+R	0.071	0.118	0.076	0.156	0.194
	TraceScore	0.087	0.134	0.079	0.187	0.241
	TraceScore+R	0.090	0.142	**0.086**	0.194	0.250
	TraceScore+W	0.091	0.136	0.077	0.196	0.250
	TraceScore+W+R	**0.093**	**0.143**	0.083	**0.203**	**0.255**

TABLE IV
PEARSON CORRELATION BETWEEN BUG LOCALIZATION METRICS IMPROVEMENT (MAP AND MRR) AND RENAMING REFACTORING

Technique	Metric	Correlation	P-value
SimiScore+R	MAP	0.789	0.004
	MRR	0.838	0.001
TraceScore+R	MAP	0.929	0
	MRR	0.876	0
TraceScore+W+R	MAP	0.813	0.002
	MRR	0.826	0.002

per [8]. To reveal the underlying cause, we split the dataset of Drools into five folds according to filed date, and experiment on each fold. We observed that the results on each fold are almost the same. But when leveraging longer history (more older bug reports), the results decrease. The reason could be that with longer history, older similar bug reports can introduce noise (i.e., files that were long ago changed, but are not relevant), while more recent similar bug reports are more helpful. However, when leveraging whole history (TraceScore+W), TraceScore decreased by 11.7% in MAP and 15% in MRR (comparing TraceScore with TraceScore+W). After adopting RAT, TraceScore+R decreased by 2.0% in MAP and 6.2% in MRR (comparing TraceScore+R with TraceScore+W+R). RAT does help to mitigate the noise brought by long histories by constructing more accurate code history. This conclusion also holds in the other projects. Among all the results, the best results are always achieved by leveraging RAT.

Results of Top 1, Top 5, and Top 10 are shown in Fig. 6 for easy visualization. As expected, the higher the renaming rate, the better the results of the bug localization. Overall, the results when analyzing the metrics that consider the top files are similar to MAP and MRR, in which projects with more refactorings have greater improvement, when considering longer histories, the improvement is more significant.

Answering RQ1.1: *We can conclude that bug localization based on bug report similarity indeed benefits from the use of rename refactorings. This conclusion is supported by the Pearson correlation coefficient, which indicates that the existence of rename refactorings are correlated to the improvement of MAP and MRR. The improvement is more significant when exploiting longer history.*

B. RQ1.2. Do code rename refactorings benefit bug localization that favor recently more frequently modified files?

We conduct experiments on Google's implementation of BugCache [15] to evaluate the effectiveness of integrating code rename refactoring into commit history-based bug localization. Experimental results are shown in Table V.

Overall, the results show that eight projects out of all 11 projects can benefit from code refactoring, with one project (Seam2) showing no significant difference and two projects (Drools and Weld) showing a small decline. Among all the improved eight projects, the MAP score of Izpack and Railo is slightly improved by around 4%. For the other six projects,

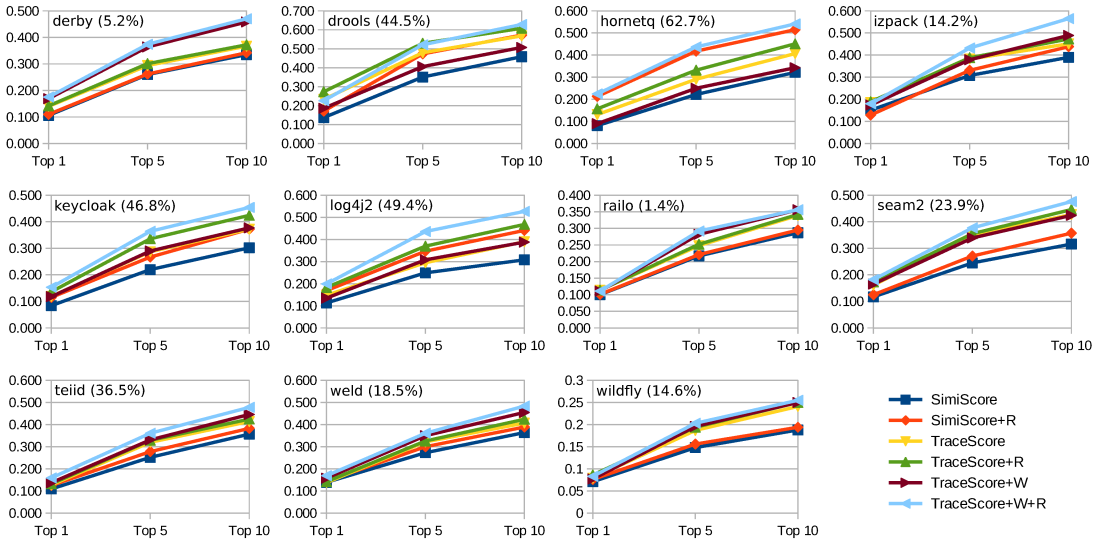

Fig. 6. Performance of SimiScore, TraceScore, and TraceScore-Whole history for Metrics Top 1, Top 5, and Top 10.

MAP scores are improved by 17%˜36%. If we look at the MRR, Top 1 and Top5 metrics, Derby and Wildfly are two that benefit the most. The Top 1 are improved by 137% and 233%, respectively. If we look at the two projects (Drools and Weld) that have lower scores when considering code refactorings, the decrease in score is almost negligible, which is less than 7%.

However, BugCache recommends recently more frequently modified files, independent of the content of bug reports, therefore the accuracy is much lower than bug report similarity. Although renamed files gain extended history (history before renaming) with RAT, their BugCache scores will not be changed a lot according to equation II-A. BugCache prefers the most recent modifications, so old modifications only contribute minor difference.

Answering RQ1.2: *For recently more frequently modified file favored bug localization, 70% projects in our dataset can benefit from code rename refactorings, with 20% of them gaining significant improvement in Top 1 (greater than 100%). 30% of projects show no significant difference.*

Existing approaches aggregate bug report similarity and version history, as well as other components by weights as indicated in Table I. The weight of bug report similarity ranges from 0.058 to 0.32 and it is from 0 to 0.4 for version history. In this way, we can see all the approaches will benefit from more complete code history, especially for approaches that adopted both with high weights.

C. RQ2. How often do code refactorings happen?

According to the first question, we can see that different bug localization approaches do benefit from code refactoring. Fig. 4 presents during the whole studied period of time, how many buggy files have been refactored (renamed). In some projects, there is a huge number of refactored buggy files, while in other projects, there are hardly any.

In this research question we want to investigate when and how often refactoring occurs (we are only concerned about the eight types mentions in Section II). Considering that commits are not uniformly distributed over a period of time. Thus, instead of calculating the occurrence of refactorings based on time, we calculate how many refactorings there are in a certain amount of commits. Specifically, for each project, we group all the commits into ten groups, and for each group, we calculate how many refactorings there are, and how many refactorings that rename a buggy file.

The results are shown in Fig 7. From the figure, we can see that refactoring happens throughout the whole history, not a rare case. This emphasizes the importance of keeping tracking of source code, because code history is being broken all the time. Sometimes refactorings may happen in bursts. For example, in the eighth group of Drools, there are 2,615 refactorings, while for the other nine groups, there are about 300 refactorings in each group. The number of refactorings in different projects varies. In Derby, there are about 10 30 refactoring for every group. While in Wildly, the average number of refactorings for each group is about 680. This variety indicates whether our code traceability model fits or not.

Answering RQ2: *Software refactoring happens throughout the software lifecycle. At some points, it can happen in bursts, which emphasizes keeping track of code history.*

VII. Threats to Validity

Threats to validity of our work come from three aspects:

- **Refactoring Mining Tool.** We leverage refactoring detection to construct our code history. Our results rely on the refactoring mining results. To mitigate this bias, we choose the state-of-the-art tool, which has the highest average precision and recall.
- **Implementation of Compared Approaches.** We implement SimiScore, TraceScore and BugCache as our

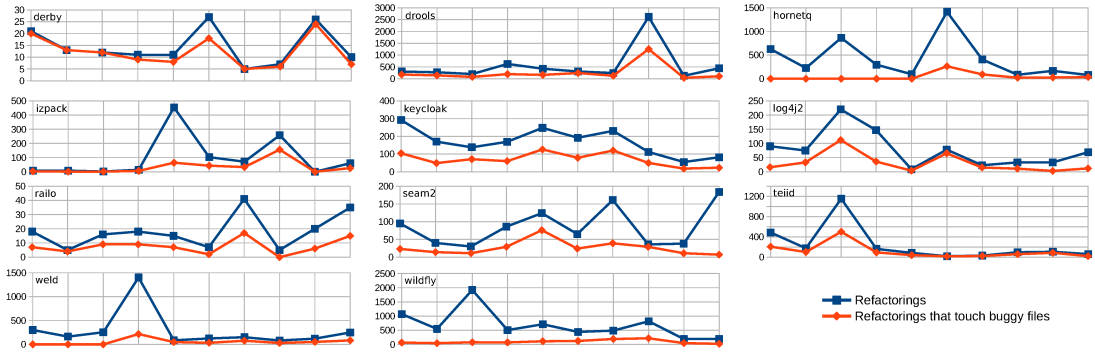

Fig. 7. Number of Refactorings and Refactorings that Touch Buggy Files in the History of Commits.

TABLE V
PERFORMANCE OF BUG CACHE WITH AND WITHOUT CONSIDERING RENAME REFACTORINGS.

Project	Approach	MAP	MRR	Top 1	Top 5	Top 10
Derby	BugCache	0.052	0.137	0.046	0.219	0.342
	BugCache+R	**0.071**	**0.194**	**0.109**	**0.282**	**0.370**
Drools	BugCache	**0.121**	**0.261**	0.165	**0.364**	**0.448**
	BugCache+R	0.118	0.253	**0.177**	0.339	0.427
Hornetq	BugCache	0.064	0.138	0.072	0.208	0.279
	BugCache+R	**0.081**	**0.162**	**0.091**	**0.219**	**0.309**
Izpack	BugCache	0.132	0.237	0.126	0.342	0.447
	BugCache+R	**0.137**	**0.265**	**0.151**	**0.377**	**0.497**
Keycloak	BugCache	0.043	0.101	0.037	0.159	0.229
	BugCache+R	**0.054**	**0.119**	**0.050**	**0.177**	**0.246**
Log4j2	BugCache	0.060	0.111	0.043	0.168	0.238
	BugCache+R	**0.072**	**0.137**	**0.068**	**0.198**	**0.268**
Railo	BugCache	0.076	0.278	0.176	0.382	0.441
	BugCache+R	**0.080**	**0.283**	0.176	0.382	0.441
Seam2	BugCache	0.047	**0.078**	**0.039**	0.114	0.151
	BugCache+R	0.047	0.077	0.037	0.114	**0.158**
Teiid	BugCache	0.034	0.087	0.031	0.129	0.201
	BugCache+R	**0.040**	**0.092**	**0.033**	**0.135**	**0.220**
Weld	BugCache	**0.083**	**0.192**	0.095	0.284	**0.384**
	BugCache+R	0.081	0.184	0.095	**0.291**	0.369
Wildfly	BugCache	0.009	0.019	0.003	0.018	0.045
	BugCache+R	**0.011**	**0.026**	**0.010**	**0.028**	**0.050**

baseline. Since there are no available packages for SimiScore and TraceScore, we re-implement their approaches strictly by their papers. For BugCache, we reused the reproducible package by Jaekwon et al. [38].

- **Refactoring Types Selection.** In this paper, we only consider about eight types of code refactoring. However, there can also be other types of refactoring that may cause influence. To reduce this threat, we only choose types that obviously rename files.

VIII. Related Work

The IR-based bug localization has been intensively studied in the literature, e.g., [6]–[8], [10]. It uses information retrieval techniques to find buggy-prone files. Zhou et al. propose BugLocator which calculate similarity between bug reports to recommend similar files to similar bug reports [6]. Sisman and Kak propose a source code version history-based fault localization approach, which utilizes the frequency of a file being buggy and its modifications to prioritize candidate source code files [11]. Wang et al. combine similar bug reports, code version history and code structure to find the buggy files [7], [10]. Wen et al. uses change logs and change hunks from commit message as alternative of segments of source code files to enable more accurate bug localization [13]. Rath et al. propose ABLoTS approach that leverages similar bug reports, requirements, code structure and commit history [8]. Jaekwon et al. conduct on a reproduction study of state-of-the-art IRBL approaches [38]. They provide their replication package [6], which makes our replication of BugCache easier. Li et al. re-implement six state-of-the-art bug localization approaches and report their effectiveness on 10 Huawei projects [39].

IX. Conclusion and Future Work

In this paper, we proposed a refactoring-aware traceability model, and based on the data model, we reconstruct the code evolution history that may have been broken by code refactoring. Then we leverage the more complete/less fragmented code history to three widely-adopted, code history-based bug localization components: SimiScore, TraceScore and BugCache. We evaluate our traceability model on 11 large scale open source projects with more than 8,000 bug reports. Experimental results show that with our traceability model, all of the existing approaches can be boosted. We also found the improvement are correlated to the refactoring amount. Therefore, we show effectiveness of considering code refactoring history into code history-based tasks. Our source code is available at https://github.com/feifeiniu-se/RAT.

In the future, we plan to improve method-level bug localization with the refactoring-aware traceability model. Also, We plan to develop broader application scenarios that leverage code history.

Acknowledgment

This work is supported by Natural Science Foundation of Jiangsu Province, China (BK20201250), Cooperation Fund of Huawei-NJU Creative Laboratory for the Next Programming, and also supported in part by NSF Grant 2034508 (USA), by a Sam Taylor Fellowship Award, the Austrian Science Fund (FWF) grant P31989-N31 and P34805-N as well as the LIT Secure and Correct System Lab sponsored by the province of Upper Austria. Jidong Ge is the corresponding author.

[6]https://github.com/exatoa/bench4bl

REFERENCES

[1] T. A. Standish, "An essay on software reuse," *IEEE Transactions on Software Engineering*, no. 5, pp. 494–497, 1984.

[2] T. A. Corbi, "Program understanding: Challenge for the 1990s," *IBM Systems Journal*, vol. 28, no. 2, pp. 294–306, 1989.

[3] M. Xie and B. Yang, "A study of the effect of imperfect debugging on software development cost," *IEEE Transactions on Software Engineering*, vol. 29, no. 5, pp. 471–473, 2003.

[4] J. Anvik, L. Hiew, and G. C. Murphy, "Coping with an open bug repository," in *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, 2005, pp. 35–39.

[5] A. Telea and L. Voinea, "Visual software analytics for the build optimization of large-scale software systems," *Computational Statistics*, vol. 26, no. 4, pp. 635–654, 2011.

[6] 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 *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 14–24.

[7] 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*, 2014, pp. 53–63.

[8] M. Rath, D. Lo, and P. Mäder, "Analyzing requirements and traceability information to improve bug localization," in *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018, pp. 442–453.

[9] K. C. Youm, J. Ahn, J. Kim, and E. Lee, "Bug localization based on code change histories and bug reports," in *2015 Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2015, pp. 190–197.

[10] S. Wang and D. Lo, "Amalgam+: Composing rich information sources for accurate bug localization," *Journal of Software: Evolution and Process*, vol. 28, no. 10, pp. 921–942, 2016.

[11] B. Sisman and A. C. Kak, "Incorporating version histories in information retrieval based bug localization," in *2012 9th IEEE working conference on mining software repositories (MSR)*. IEEE, 2012, pp. 50–59.

[12] F. Rahman, D. Posnett, A. Hindle, E. Barr, and P. Devanbu, "Bugcache for inspections: hit or miss?" in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, 2011, pp. 322–331.

[13] M. Wen, R. Wu, and S.-C. Cheung, "Locus: Locating bugs from software changes," in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2016, pp. 262–273.

[14] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller, "Predicting faults from cached history," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 489–498.

[15] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead, "Does bug prediction support human developers? findings from a google case study," in *2013 35th International Conference on Software Engineering (ICSE)*. IEEE, 2013, pp. 372–381.

[16] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2013, pp. 345–355.

[17] A. Marcus and J. I. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *25th International Conference on Software Engineering, 2003. Proceedings.* IEEE, 2003, pp. 125–135.

[18] 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 *2014 IEEE international conference on software maintenance and evolution*. IEEE, 2014, pp. 181–190.

[19] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, "Refactoring: Improving the design of existing code addison-wesley professional," *Berkeley, CA, USA*, 1999.

[20] M. Fowler, *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 2018.

[21] G. Sellitto, E. Iannone, Z. Codabux, V. Lenarduzzi, A. De Lucia, F. Palomba, and F. Ferrucci, "Toward understanding the impact of refactoring on program comprehension," in *29th International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2022, pp. 1–12.

[22] A. Brito, A. Hora, and M. T. Valente, "Refactoring graphs: Assessing refactoring over time," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 367–377.

[23] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 313–324.

[24] D. Silva and M. T. Valente, "Refdiff: detecting refactorings in version histories," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 269–279.

[25] D. Silva, J. Silva, G. J. D. S. Santos, R. Terra, and M. T. O. Valente, "Refdiff 2.0: A multi-language refactoring detection tool," *IEEE Transactions on Software Engineering*, 2020.

[26] N. Tsantalis, A. Ketkar, and D. Dig, "Refactoringminer 2.0," *IEEE Transactions on Software Engineering*, 2020.

[27] N. Tsantalis, M. Mansouri, L. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 483–494.

[28] M. Di Penta, G. Bavota, and F. Zampetti, "On the relationship between refactoring actions and bugs: a differentiated replication," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, pp. 556–567.

[29] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? an empirical study," in *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*. IEEE, 2012, pp. 104–113.

[30] R. Halepmollasi and A. Tosun, "Exploring the relationship between refactoring and code debt indicators," *Journal of Software: Evolution and Process*, p. e2447, 2022.

[31] A. Brito, A. C. Hora, and M. T. Valente, "Characterizing refactoring graphs in java and javascript projects," *Empirical Software Engineering*, vol. 26, no. 6, pp. 1–43, 2021.

[32] G. Lacerda, F. Petrillo, M. Pimenta, and Y. G. Guéhéneuc, "Code smells and refactoring: A tertiary systematic review of challenges and observations," *Journal of Systems and Software*, vol. 167, p. 110610, 2020.

[33] R. O. Chris Lewis, "Bug prediction at google," [EB/OL], http://google-engtools.blogspot.com/2011/12/bug-prediction-at-google.html Accessed 12, 2011.

[34] K. C. Youm, J. Ahn, and E. Lee, "Improved bug localization based on code change histories and bug reports," *Information and Software Technology*, vol. 82, pp. 177–192, 2017.

[35] 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 *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2015, pp. 476–481.

[36] X. Ye, R. C. 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*, 2014, pp. 689–699.

[37] S. Wang, D. Lo, and J. Lawall, "Compositional vector space models for improved bug localization," in *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 2014, pp. 171–180.

[38] J. Lee, D. Kim, T. F. Bissyandé, W. Jung, and Y. L. Traon, "Bench4bl: Reproducibility study of the performance of ir-based bug localization," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2018, 2018, pp. 1–12.

[39] W. Li, Q. Li, Y. Ming, W. Dai, S. Ying, and M. Yuan, "An empirical study of the effectiveness of ir-based bug localization for large-scale industrial projects," *Empirical Software Engineering*, vol. 27, no. 2, pp. 1–31, 2022.