The ABLoTS Approach for Bug Localization: is it replicable and generalizable?

Feifei Niu*, Christoph Mayr-Dorn†, Wesley K. G. Assunção†, LiGuo Huang‡,
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: {christoph.mayrdorn, wesley.assuncao, alexander.egyed}@jku.at
‡Department of Computer Science and Engineering, Southern Methodist University, Dallas, Texas, USA
Email: lghuang@lyle.smu.edu

***Abstract*—Bug localization is the task of recommending source code locations (typically files) that probably contain the cause of a bug and hence need to be changed to fix the bug. Along these lines, information retrieval-based bug localization (IRBL) approaches have been adopted, which identify the most bug-prone files from the source code space. In current practice, a series of state-of-the-art IRBL techniques leverage the combination of different components, e.g., similar reports, version history, code structure, to achieve better performance. ABLoTS is a recently proposed approach with the core component, TraceScore, that utilizes requirements and traceability information between different issue reports, i.e., feature requests and bug reports, to identify buggy source code snippets with promising results. To evaluate the accuracy of these results and obtain additional insights into the practical applicability of ABLoTS, supporting of future more efficient and rapid replication and comparison, we conducted a replication study of this approach with the original data set and also on an extended data set. The extended data set includes 16 more projects comprising 25,893 bug reports and corresponding source code commits. While we find that the TraceScore component as the core of ABLoTS produces comparable results with the extended data set, we also find that the ABLoTS approach no longer achieves promising results, due to an overlooked side effect of incorrectly choosing a cut-off date that led to training data leaking into test data with significant effects on performance.**

***Index Terms*—bug localization, information retrieval, replication study**

I. Introduction

A software bug refers to an error, fault, or flaw that produces unexpected results or causes a system to behave unexpectedly [1]. A bug may cause the system to crash or become vulnerable to security attacks [2], [3]. Bugs are a common phenomenon. For example, a Mozilla triager complained that "every day, almost 300 bugs appear that need triage" [4]. Considering the severe consequences and frequent occurrences, bugs need to be responded to promptly and coped seriously. To this end, various techniques to assist this process have been suggested, for example, defect prediction [5], [6], bug triaging [7], [8], bug fixing [9], [10], and bug localization [11], [12].

Bug localization is one of the main challenges when solving bugs, which is identifying the parts of source code that cause the bug and need to be changed in order to fix it [13]. However, finding the buggy files from the source code can become a daunting task [14], especially in large projects consisting of thousands of source code files. To help to deal with this issue, several researchers proposed automatic approaches for bug localization [14]–[17].

Among existing approaches for bug localization, there is a series of them that leverage bug reports for better localization [14], [16], [17], since bug reports often contain rich information that allows us to infer the bug's location. Approaches that utilize the textual content of bug reports are generally described as information retrieval-based bug localization (IRBL). For a given bug report, IRBL tries to find and rank code snippets that may be relevant to the bug report [18], which is usually done by calculating the similarity between the bug report and source code [18]. For example, Saha et al. [19] propose the BLUiR approach that extracts structured information (e.g., class names, method names, variable names, and comments) from source code and calculates the textual similarity between the source code and bug reports to retrieve buggy files. However, there exists a lexical gap between bug reports and source code files [20]. The terms used to describe the bug in the bug report may not match the terms used in class names, methods names, variable names, or comments. Not surprisingly, textual similarity by itself will not necessarily yield good results [14].

To this end, state-of-the-art approaches leverage multiple sources of information to improve the performance of bug localization. Wang et al. propose the AmaLgam approach, which combines code structure, similar bug reports, and version history [14]. BRTracer+ leverages bug reports similarity and stack trace from bug reports for bug localization [21]. Youm et al. integrate stack trace information with all those pieces of information used by AmaLgam [22]. AmaLgam+ leverages five sources of information, namely version history, similar bug reports, code structure, stack trace, and reporter information [17].

Recently, Rath et al. presented a new approach, named ABLoTS, that leverages not only similar bug reports, version history, code structure, but also similar non-bug reports, like feature requests, enhancements, tasks, and so on, as well

as traceability information between bug reports and other types of issues [23]. Rath et al. reused the structure of AmaLgam, but proposed TraceScore to replace the similar bug reports component, and additionally decided to use a decision tree (DT) for dynamically combining the recommendations from the individual components. The experimental evaluation showed that ABLoTS greatly outperforms AmaLgam.

Although the original study by Rath et al. [23] showed encouraging results (with no other state-of-the-art approaches exhibiting better performance [24], [25]), there are no replications in the literature that confirm its outstanding performance. Additionally, there are no studies that investigate whether the performance also holds for a larger data set, i.e., that evaluate the generalization of ABLoTS. A replication study is helpful and necessary to verify experimental results from previous studies [26]. They are a key aspect of empirical software engineering, as they bring evidence that observations made can hold (or not) under other conditions [27]. Extensive and independent evaluations are also necessary to reach industrial adoption and practice [28], [29].

In this paper, we present a literal and conceptual replication [30] of the ABLoTS approach. We replicate the experiments as closely as possible to the initial procedures. Meanwhile, we also run the experiment on another new data set without changing anything else, to see how well the results hold up. To this end, we first re-implemented TraceScore, the core component of ABLoTS, and checked the replicability of the results on the original data set. Then, we replicated the overall ABLoTS framework on the original data set. Additionally, we investigated the TraceScore's and ABLoTS' generalizability based on an extended data set from Rath and Mäder [31]. Our work is organized according to standard replication report guidelines for software engineering studies [27]. This is an external and independent replication study without any of the authors of the original paper taking part in the replication process.

In general, our replication results show that TraceScore is replicable and generalizable under specific settings. However, ABLoTS is neither replicable on the original data set nor on a larger data set [31]. Specifically, we observed that the implementation of ABLoTS reused a subcomponent from prior work (AmaLgam [14]) that incorrectly sets a cut-off date, which leads to training data leaking into test data.

The contributions of this paper are:

1) an empirical investigation showing that the TraceScore component is replicable and generalizable, thus strengthening confidence that relations between bug reports and feature requests yield useful information for bug localization.
2) a failed attempt to replicate the promising results of the ABLoTS approach, thereby showing that bug localization still needs significant research efforts and is not ready for practical application.
3) identification of the major reason why replication failed, thereby highlighting the challenge of reusing research results.
4) a lab package[1] to replicate our experiment and evaluate the ABLoTS approach.

The remainder of this paper is organized as follows. Section II summarizes the original study, approach, evaluation, and achieved results. Section III elaborates our replication study design, research questions, and data set. The experimental results are presented and discussed in Section IV. Section V discusses threats to validity. Related work is presented in Section VI before Section VII concludes this work.

II. Original Study

In this section, we provide an overview of the bug localization technique by Rath et al. [23]. We firstly present the TraceScore component that is at the center of Rath et al. ABLoTS approach, encapsulated in the **Similar Reports Component** (see Fig 1) (Section II-A). Then, we present the whole framework of the ABLoTS approach (Section II-B), the utilized evaluation metrics (Section II-C) as well as the data set as used in the original study (Section II-D). Finally, we summarize the reported experimental results (Section II-E).

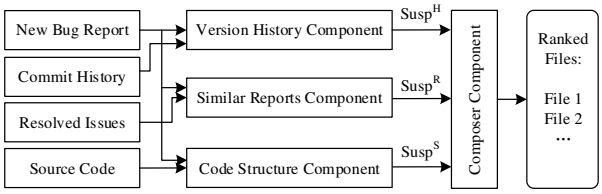

Fig. 1. Components of ABLoTS.

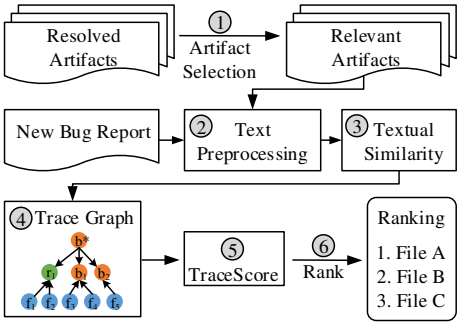

Fig. 2. TraceScore Component.

A. TraceScore Component

TraceScore is one of the main components of the ABLoTS approach. Specifically, it introduces a novel calculation scheme for the **Similar Reports Component**. The core idea is that similar bugs will be caused by similar source code snippets. Hence, by identifying similar bugs and inspecting which files were changed in their bug-fixing commits, one can obtain a list of files indicating the bug location.

TraceScore mainly consists of six steps, as shown in Fig 2. It takes a new bug report $b*$, previously resolved bug reports B and non-bug reports R (e.g., feature request and enhancements)

[1] https://github.com/feifeiniu-se/Replication

as input. **Step 1** is artifact selection, based on two criteria, i.e., *time domain* and *number of modified files*. For the *time domain*, bug reports $b \in B$ and feature requests $r \in R$ that are fixed within "one year before $b*$ was filed" to "the date when $b*$ was filed", would be retained. As for *number of modified files*, only bug reports $b \in B$ that modify no more than 10 Java files and feature requests $r \in R$ that modify no more than 20 Java files will be retained. The reasons for adoption of these two criteria and their validity are explained in Section 6 of the original study. **Step 2** utilizes commonly used preprocessing techniques to build a document-term-matrix [32] of the filtered artifacts from Step 1. Then, TraceScore calculates the cosine similarity between $b*$ and each artifact in **Step 3**. In **Step 4**, a trace graph is created, with $b*$ as the root node, linked to sub-graphs of different artifacts, by the edges indicating textual similarity between $b*$ and each artifact (if there is a trace link between $b*$ and artifact, the edge is set to 1). Each artifact traces further to the files that are part of a corresponding commit in the version control system. In this way, b* is indirectly linked to a potentially large set of source code files, that need subsequent ranking, where the ranking happens on the basis of a **TraceScore** between each file and $b*$ which is calculated by (1) in **Step 5**. Finally, **Step 6** sorts all the source code files linked to $b*$ according to TraceScore and outputs the ranked list. A higher score indicates a higher likelihood of that file being relevant.

$$Susp^R(s, b*) = \sum_{a_i \in \{a | s \in fix(a)\}} \frac{sim(a_i, b*)^2}{|fix(a_i)|} \quad (1)$$

B. ABLoTS Approach

The overall ABLoTS approach consists of four components: 1) similar reports component, 2) version history component, 3) code structure component, and 4) composer component, as shown in Fig 1. TraceScore is an implementation of a similar reports component. We only briefly describe the other three components, as these are reused by Rath et al. without changes.

Version History Component makes use of BugCache [5], [33], to predict which files are likely to be buggy in the future. BugCache takes commit history as input and outputs a list of files with a high "suspiciousness" score. To this end, it firstly identifies bug-fixing commits (commits whose commit messages contain the word "fix" or "bug") that were committed within k days prior to the submission of the new bug report $b*$. Then the suspiciousness score of each file f is calculated by (2), where f is one of the buggy files in commit $c \in C$, t_c is the elapsed time in days between the commit c and when the bug report was filed. k was set to 15 (days) according to Wang et al. [14].

$$Susp^H(f, b*) = \sum_{c \in C \wedge f \in c} \frac{1}{1 + e^{12(1-((k-t_c)/k))}} \quad (2)$$

Code Structure Component leverages BLUiR [19] to identify files from source code space according to the similarity between source code files and bug report $b*$. It outputs a ranked list of files with a suspiciousness score $Susp^S(f, b*)$.

Composer Component aggregates the three suspiciousness scores obtained by the first three components, i.e., $Susp^R$, $Susp^H$, $Susp^S$, and outputs the final results. Instead of adopting a fixed weight scheme for the three scores as done by Wang et al. [14], [17], ABLoTS applied Weka's [34] J48 DT to learn the best combination. For training, the classification algorithm takes $Susp^R(f, b*)$, $Susp^H(f, b*)$, $Susp^S(f, b*)$ as the features, and whether that file f was changed as part of the bug fix or not as the classification result. For each project separately, they trained the classifier on 80% of the bug reports that were resolved and evaluated ABLoTS on the remaining 20% that were resolved after the 80% cut-off deadline.

C. Evaluation Metrics

To evaluate the effectiveness of the approach, Rath et al. adopted the following commonly used metrics:

Top@k [35] measures the percentage of bug reports in which at least one of the buggy files is in top *k* ranked files, where *k*=1,5,10.

Mean Average Precision (MAP) [32] 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:

$$AP = \sum_{i=1}^{N} \frac{P(i) * pos(i)}{\#\, of\, positive\, instances} \quad (3)$$

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.

Mean Reciprocal Rank (MRR) [36] computes the average of the reciprocal of the positions of the first correctly located buggy file in the ranked files.

D. Original Data Set

In the original study, Rath et al. contributed a data set [37] consisting of 15 popular open-source projects with 13,581 bug reports and 9,219 feature requests. Firstly, they collected issues (i.e., bug reports and feature requests), as well as the dependency trace links from Jira [38], and downloaded source code of these projects from GitHub [39]. Then the heuristic proposed in [40] was applied to create links between issues and commits. The ABLoTS approach was evaluated based on this data set.

E. Achieved Performance Originally Reported

The achieved performance by Rath et al. is shown in Table I, which is the average of 15 projects. According to Rath et al., TraceScore benefits from leveraging non-bug issues as well as traceability information. It can outperform two state-of-the-art similar reports based approaches: SimiScore [16] and CollabScore [41]. The overall ABLoTS framework outperforms the AmaLgam framework [14] which was used as a baseline.

TABLE I
ORIGINAL, REPORTED PERFORMANCES [23]

Algorithm	MAP	MRR	Top 1	Top 5	Top 10
TraceScore	0.202	0.260	0.174	0.350	0.436
ABLoTS	0.488	0.545	0.487	0.610	0.649

III. REPLICATION METHODOLOGY

Our goal of the replication study is to investigate whether the results based on the TraceScore component and ABLoTS approach are replicable and generalizable. To this end, we 1) replicate the component and the approach on a subset of the original data set and 2) apply the component and the approach on an extended data set consisting of 16 more projects.

This study is considered to be an external [27] replication study of the original study, since none of the authors took part in the replication process. However, we reused 11 projects of the original data set to verify the results [2].

A. Research Questions

In the scope of this paper, we aim at answering the following two research questions:

- RQ1. How effective is TraceScore in identifying bug-relevant source code files?
  - RQ1.1 Are we able to replicate the original performance of the TraceScore component?
  - RQ1.2 Does the TraceScore component yield similar performance when applied to other data?
- RQ2. How effective is ABLoTS for bug localization?
  - RQ2.1 Are we able to replicate the results of the ABLoTS approach?
  - RQ2.2 Does the ABLoTS approach yield similar performance when applied to other data?

Our research questions are adapted from those addressed in the original study, which involve the main contribution of the Rath et al. study [23]. Specifically, RQ1 is adapted from RQ1 of the original study, which evaluates the TraceScore component. RQ2 is adapted from RQ4 of the original study, which evaluates the ABLoTS approach. For each research question, we replicate from two dimensions, i.e., replicability and generalizability. The core difference of the dimensions is the data set being used for evaluation. For the replicability validation, we evaluate on the same projects with the original study, to see if our replication results are consistent with the original results. As for the generalizability validation, we adopted an extended data set to see if the approach is applicable to other projects as well. The other two RQs of the original study mainly investigate the effectiveness of artifacts selection, which is irrelevant of our goal, so we do not include them in this study.

[2]The other four projects from the original data set were excluded due to some missing commits on GitHub.

B. RQ1. How effective is TraceScore in identifying bug-relevant source code files?

This research question mainly focuses on the main contribution of Rath et al., i.e., TraceScore for the similar reports component.

As the original source code is not available, we followed the procedures proposed in the original paper (as illustrated in Section II-A) as close as possible to duplicate all facets of TraceScore. Specifically, on each project basis, we sort all the issues according to the resolved date. Then we split all the bug reports 80:20, with the latter 20% used as the test set to recommend buggy files.

As in the original study, we filter the number of related bug reports and features as well as commits based on age and size from which to obtain a recommendation. For each bug report $b*$ in the test set, we consider only bug reports (and features) b that occurred before $b*$ as determined by the following condition: "$b.fixed\_date > b*.created\_date - 365\,days$". However, we are of the opinion that there is another constraint that also should be satisfied: "$b.fixed\_date < b*.created\_date$", which means that only bug reports fixed before $b*$ were filed should be retained. These two settings describe the following two recommendation situations: the former describes the bug localization mechanism called shortly before fixing the bug, close to the bug report's closing date, while the latter describes a recommendation immediately made upon bug creation. For our replication, we were unable to determine whether the authors only adopted the first constraint (denoted as *relaxed cut-off date*) or adopted both constraints (denoted as *strict cut-off date*). We conducted the replication with both relaxed and strict cut-off date to understand the impact the additional constraint has on the results. Then, we select bug reports/features according to the *number of modified files* identified in their commits. We exclude issues that modify more than 10 files for bug reports and more than 20 files for non-bug reports). We then build up the trace graph from these issue subset as shown in Fig 2 in Step 4. The edges between the root node $b*$ and other artifacts are calculated using cosine similarity [42]. When an issue explicitly links to another issue, then the link weight overrides the cosine similarity and becomes 1. With the trace graph, the TraceScore between each file node s and $b*$, $TraceScore(s, b*)$ is calculated. Finally, all the files according to their tracescore, we will get the ranked list for $b*$.

Then we evaluate our replication on the extended data set. For the extended data set, we applied preprocessing as **Step 2** (in Section II-A) to be consistent with the original data set and to fit the replication. Specifically, for each issue, we preprocessed the text including both summary and description according to **Step 2** in Section II-A. We utilize NLTK library [43] in Python for preprocessing, including stop words removing, camel case splitting, lower casing and stemming. Then the preprocessed texts are converted into TF-IDF [44] vector with the sklearn library [45]. For the source code, we exclude non-source code files based on the file name extension

and only retain Java files ("*.java"). For each file changed in each commit, the extended data set contains the old name and the new name for this file. According to the original study, we only utilize the new name for each file, which means, removed files will be excluded for each commit.

C. RQ2. How effective is ABLoTS for bug localization?

The ABLoTS approach is essentially an ensemble of three components, i.e., similar reports, version history, and code structure, as shown in Fig 1. Each strategy outputs a suspiciousness score for a given bug report $b*$ and source code file s, denoted by $Susp^R(s, b*)$, $Susp^H(s, b*)$, and $Susp^S(s, b*)$. Then a composer component aggregates all three scores to determine the final classification result. As described in the original study, the ABLoTS approach is an evolved version of AmaLgam [14] with two main differences: first, it replaces the similar reports component with TraceScore; second, it applies a dynamic suspiciousness score combination (instead of the former static one). At the time of conducting the replication, there is no open source code available for the whole framework. We, therefore, replicated the overall framework along the following lines.

Version History. As mentioned in Section II-B, the version history component is implemented by BugCache, which is proposed by Kim et al. [5]. BugCache maintains the modification history of files to predict buggy-prone files in the future. It proved that more recently and frequently modified files are more likely to be buggy in the future. Rahman et al. proposed a simpler version of BugCache [33], which only maintains a short history of file modification. Google's developers adapted Rahman et al.'s algorithm on their large systems [46], [47]. AmaLgam adapted Google's well-tested algorithm with a version history component. We reused AmaLgam's implementation of BugCache[3], but made the following modifications:

1) The BugCache version used in AmaLgam was written in Java, while we manually translated it to Python, to be compatible with our implementation.

2) In their paper, Wang et al. [14] elaborate that the approach identifies commits that are committed 15 days before *the new bug report is created*. However, after checking the source code, we found that the implementation utilized the bug report's *resolved date* as the cut-off date to obtain previously committed commits within 15 days. We contacted the authors, and they agreed that the bug report's *creation dates* should have been adopted. Therefore, in our implementation, we used *the creation date* for all our experiments.

3) To identify bug-fixing commits, Wang et al. proposed that commit logs should match regular expression regex: *(.*fix.*)|(.*bug.*)*. Considering that some programming languages (e.g., Java and Python) are case-sensitive, we firstly convert commit logs into lowercase, which is missing in the original implementation. What is more, according to our observation of the data set, some bug-fixing commit logs maybe not contain keywords like "fix" or "bug". However, they might start with the bug report's ID. To this end, we also include commits that start with any bug ID in their logs, to identify bug-fixing commits more accurately. AmaLgam's authors also agree with us on this. This adapted selection of commits only affects the commits used for BugCache, but not any other component in ABLoTS.

Code Structure. Code Structure metrics are obtained with BLUiR [19], which calculates the similarity between a new bug report and the code structure of a source code file. It takes summary and description of a bug report as two separate parts and extracts class names, method names, variable names, and comments of a source code file with Abstract Syntax Tree. Then it indexes and searches buggy files based on the Indri toolkit [48]. In this paper, instead of replicating our own BLUiR tool, we used the implementation[4] without any modification from an empirical study by Lee et al. [24] to obtain the $Susp^S(s, b*)$ score.

Composer. ABLoTS applied J48 DT with default pruning settings to classify source code files for bug reports. Specifically, for each $b*$, there are multiple candidate source code files s for recommendation. For each (s,b*), there will be a label $C \in \{true, false\}$ indicating whether the file s is modified to fix b or not. For training, the classifier takes the $Susp^R(s, b*)$, $Susp^H(s, b*)$, and $Susp^S(s, b*)$ scores for each (s, b*) as feature and C as label. For test data, instead of output a label indicating true or false, the probability of s being *true* (i.e., s is modified by $b*$) is utilized. Then for each $b*$, all the files are ranked according to the probability score.

On each project basis, Rath et al. sorted all the bug reports by resolved date and took the first 80% bug reports as training data, and the rest 20% as test data. To mitigate the influence of imbalanced training data, ABLoTS used Weka's sub-sampling to under-sampling the training data.

Since our replication is based on Python, we chose the popular open sourced Python library sklearn [45] for the *DT* classifier and *RandomUnderSampler* in the Imblearn library [49] for under-sampling. Essentially they are the same algorithm with the original study, but just implemented by different libraries. We assume that this will not cause significant difference to the result as we used exactly the same training data as in the original paper (i.e., rather than sampling our own set of training data we utilized the precalculated suspiciousness scores and classification result from the replication package to obtain a trained DT).

We applied the same procedure on the original data set and the extended data set.

D. Data set

For the replicability validation, we reuse the data set provided for replication by Rath et al. [37]. However, by the time we carried out the replication study, many commits from four projects (namely, Axis2, Hadoop, Infinispan, and Pig) were no longer available programmatically on GitHub and neither are

[3]https://sites.google.com/view/mambalab/projects/amalgam

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

part of the original replication package. Hence, as we could not obtain complete commit history for BugCache, we excluded these four projects from the analysis in this paper.

In order to investigate the generalizability of TraceScore component and ABLoTS approach, we picked an extended data set, SEOSS 33 data set [31], which includes an additional 18 projects and 36,482 bug reports out of which we could not use 2 projects due to the same issue of non-accessible commits. This extended data set also includes the 15 original projects from the replication package [37]. Details about the extended data set are shown in Table II. We choose this data set because it not only links bug reports to commit code change, but also includes traceability information between bug reports and non-bug issues, which caters to our needs perfectly.

Apart from the information in the data set, we additionally collected version information for each project. In each commit, developers may modify a file, add a new file, or remove an old file. Removed files are obsolete and should not appear in the recommendation of a new bug report. However, the similar reports component leverages historical issues, which may be pointing to no longer existing files. In this way, they may bias the prediction results. To address this issue, we determine for each commit which files exist just prior to this commit. Files in this set can only be used as the candidates to recommend the bug's location.

TABLE II
CHARACTERISTICS OF THE EXTENDED DATA SET.

PROJECTS	Time Period (Month)	# Bug Reports	# Non-bug Reports	# Commits
ARCHIVA	162	371	411	8006
CASSANDRA	106	3571	2813	23592
ERRAI	99	267	194	7645
FLINK	43	1350	2351	12419
GROOVY	173	1933	1017	12378
HBASE	131	4581	5171	14331
HIBERNATE	172	1947	1706	8173
HIVE	113	4776	4326	11179
JBOSS-T.-M.	145	331	489	2204
KAFKA	78	639	1149	4426
LUCENE	197	3773	5324	28995
MAVEN	183	760	574	10315
RESTEASY	119	345	228	3684
SPARK	93	328	7022	20829
SWITCHYARD	86	451	759	2928
ZOOKEEPER	116	470	471	1600

IV. RESULTS AND DISCUSSION

A. RQ1. How effective is TraceScore in identifying bug-relevant source code files?

RQ1.1 Replicability. We carried out the replication according to Section III-B. Results on the original data set are as shown in Table III. The performance impact of using the *strict cut-off date* is on average around 17% lower than using the *relaxed cut-off date*.

To find out which implementation most likely has been adopted by the original implementation, we performed a pairwise t-test on the 11 projects, comparing both replication results against the reported results in [23] to establish statistically whether these results can be considered to be the same. According to the pairwise t-test, the *relaxed cut-off date* is closer to the original implementation. The pairwise t-test results (as shown in Table IV) show that for MRR, Top1, Top5, and Top10, there is no significant difference while for MAP we have to reject the null hypothesis for the *relaxed cut-off date*: the average MAP reported by Rath et al. is 32% higher than our replication result. For the remaining four evaluation metrics, there is no significant difference; the mean values are statistically the same. So we conclude that with the *relaxed cut-off date* TraceScore can be considered replicable while with the *strict cut-off date* it cannot be considered replicable as we cannot achieve statistically comparable or better results.

To give benefit to doubt, we adopted the *relaxed cut-off date* for the remainder of the replication and generalization investigations. However, in practice, the choice between *relaxed cut-off date* and *strict cut-off date* is artificial as only commits available at the time the bug localization mechanism is applied are considered for producing the recommendation.

RQ1.2 Generalizability The evaluation results based on the extended data set are shown in Table V. The average MAP, MRR, Top 1, Top 5 and Top 10 are 18.3%, 28.4%, 19.6%, 38.4%, 47.3%, respectively. The MAP value is distributed between 4.4% and 32.5%. In order to confirm if there is a difference between the distribution of the original results and extended results, we leverage the two-sample Kolmogorov-Smirnov test (K-S test) [50], which is used to test whether two samples come from the same underlying one-dimensional probability distribution. For each evaluation metric, we perform a two-sample K-S test, with one sample being the results from the original data set and the other sample being the results from the extended data set. Results are shown in the "TraceScore" column of Table VI. Since all the p-values are larger than 5%, we can assert that the two samples come from the same distribution. The left plot in Fig 3 shows the data value on all five metrics, from which we can see that on the extended data set, TraceScore yields slightly higher median and wider variations. The average over the extended data set is about 12% $\sim$ 27% higher than on the original data set (e.g., MAP is 26% higher). But there is no major difference overall.

Moreover, we also investigate the improvement of TraceScore over the same baseline as in the original paper. With SimiScore [16] as baseline, we obtain the improvement of TraceScore over SimiScore on both original and extended data set. The "Improvement" column of Table VI shows the results of the K-S test. Given the p values, the improvement of MRR, Top 1, and Top 10 on the original data set and the extended data set are very likely to come from different distributions. To this end, from the middle box plot in Fig 3 for improvement, we can observe a much higher median, maximum, and minimum, which indicates TraceScore yields higher performance improvement on the extended data set.

We can therefore conclude that the performance of TraceScore also holds for a larger data set, and we gain confidence that TraceScore's performance is generally achievable.

TABLE III
TRACESCORE PERFORMANCE ON THE ORIGINAL DATA SET.

Relaxed Cut-off Date					
PROJECTS	MAP	MRR	Top 1	Top 5	Top 10
DERBY	0.124	0.240	0.149	0.340	0.404
DROOLS	0.183	0.383	0.276	0.502	0.615
HORNETQ	0.134	0.241	0.130	0.352	0.481
IZPACK	0.170	0.229	0.156	0.328	0.422
KEYCLOAK	0.125	0.234	0.152	0.323	0.418
LOG4J2	0.182	0.271	0.191	0.360	0.416
RAILO	0.138	0.202	0.117	0.267	0.350
SEAM2	0.134	0.195	0.141	0.244	0.288
TEIID	0.194	0.278	0.188	0.385	0.465
WELD	0.102	0.208	0.098	0.312	0.420
WILDFLY	0.108	0.185	0.116	0.268	0.326
Average	**0.145**	**0.242**	**0.156**	**0.335**	**0.419**

Strict Cut-off Date					
PROJECTS	MAP	MRR	Top 1	Top 5	Top 10
DERBY	0.084	0.158	0.096	0.219	0.272
DROOLS	0.171	0.37	0.265	0.467	0.603
HORNETQ	0.105	0.207	0.093	0.315	0.444
IZPACK	0.101	0.152	0.094	0.219	0.297
KEYCLOAK	0.081	0.16	0.082	0.241	0.323
LOG4J2	0.165	0.256	0.18	0.315	0.382
RAILO	0.131	0.194	0.117	0.25	0.35
SEAM2	0.099	0.159	0.103	0.212	0.263
TEIID	0.140	0.222	0.135	0.331	0.412
WELD	0.103	0.201	0.098	0.304	0.411
WILDFLY	0.085	0.146	0.087	0.217	0.268
Average	**0.115**	**0.202**	**0.123**	**0.281**	**0.366**

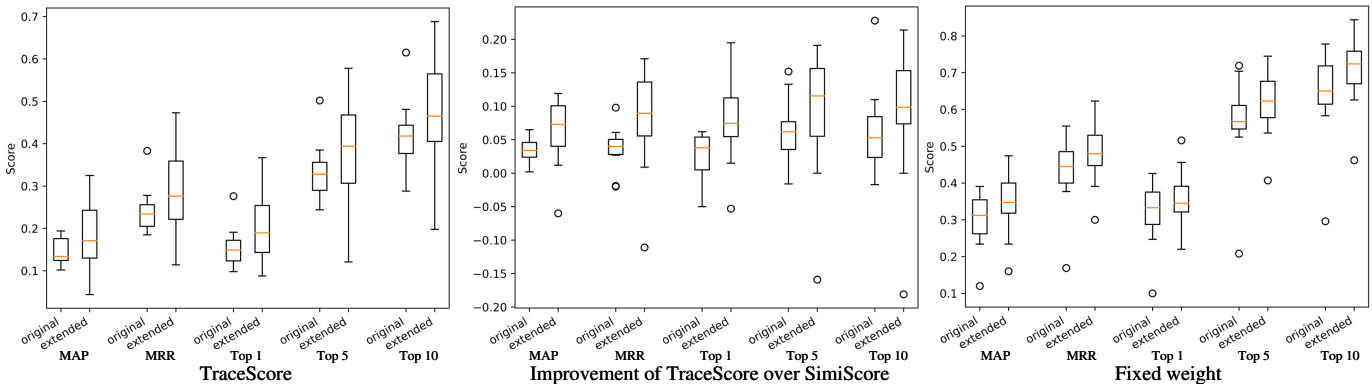

Fig. 3. Box plots of TraceScore, improvement of Tracescore and fixed weight, on both original and extended data set.

TABLE IV
PAIRWISE T-TEST BETWEEN RELAX CONSTRAINT RESULT AND ORIGINAL RESULT.

Metrics	Pairs		Deviation	P value
	Original	Replication		
MAP	0.191	0.145	0.05	0.000**
MRR	0.248	0.242	0.01	0.522
Top 1	0.163	0.156	0.01	0.407
Top 5	0.336	0.335	0.00	0.924
Top 10	0.419	0.419	0.00	0.99

*p <0.05 **p <0.01

Answering RQ1: *Under the relax cut-off date constraint, TraceScore is replicable and also can be generalized to an extended data set. However, under the strict cut-off date constraint, we cannot claim replicability as the performance is significantly lower than reported.*

B. RQ2. How effective is ABLoTS for bug localization?

RQ2.1 Replicability ABLoTS's performance results on the original data set are shown in Table VII. Compared to the results reported in the original paper (cf. Table I) we note that our replication produces far worse results. MAP and MRR are below 10% for most projects. ABLoTS, which combines three scores: $Susp^R$, $Susp^H$, and $Susp^S$, does not even achieve the same results as the single $Susp^R$ score. This counterintuitive

TABLE V
TRACESCORE PERFORMANCE ON THE EXTENDED DATA SET.

PROJECTS	MAP	MRR	TOP 1	Top 5	Top 10
ARCHIVA	0.134	0.22	0.147	0.28	0.413
CASSANDRA	0.218	0.333	0.222	0.453	0.551
ERRAI	0.059	0.15	0.093	0.204	0.296
FLINK	0.18	0.305	0.207	0.415	0.522
GROOVY	0.325	0.393	0.271	0.522	0.625
HBASE	0.236	0.352	0.25	0.455	0.561
HIBERNATE	0.118	0.231	0.172	0.3	0.359
HIVE	0.264	0.38	0.267	0.506	0.599
JBOSS-T.-M.	0.136	0.247	0.164	0.373	0.433
KAFKA	0.296	0.473	0.367	0.578	0.688
LUCENE	0.201	0.32	0.228	0.419	0.494
MAVEN	0.162	0.222	0.132	0.316	0.382
RESTEASY	0.101	0.202	0.101	0.348	0.435
SPARK	0.31	0.383	0.273	0.545	0.576
SWITCHYARD	0.044	0.114	0.088	0.121	0.198
ZOOKEEPER	0.149	0.226	0.149	0.309	0.436
Average	**0.183**	**0.284**	**0.196**	**0.384**	**0.473**

TABLE VI
K-S TEST RESULT.

Metrics	TraceScore		Improvement		Fixed Weight	
	K-S test	P value	K-S test	P value	K-S test	P value
MAP	0.438	0.124	0.500	0.054	0.313	0.452
MRR	0.409	0.175	0.693	0.002	0.295	0.512
Top 1	0.409	0.175	0.625	0.007	0.210	0.856
Top 5	0.409	0.175	0.443	0.115	0.443	0.115
Top 10	0.415	0.159	0.540	0.028	0.358	0.289

result motivated us to investigate in more detail how this outcome can be explained.

For the strict replication, we trained the DT on the intermediate three scores (i.e., $Susp^R$, $Susp^H$, $Susp^S$) made available by Rath et al. in their replication package, For comparison, we also trained a separate DT from our own sample of files, their suspiciousness scores, and bug reports. Note that the original replication package just provided tuples of suspiciousness scores and classification results, but not which bug report and which files were used to obtain those suspiciousness scores. We, however, applied the same sampling criteria.

We inspected the original DT (i.e., the one obtained from the replication data) to obtain the average[5] feature importance (non-normalized) of each component: 0.037 for BLUiR, 0.377 for BugCache, and 0.018 for TraceScore. This indicates that BugCache almost exclusively determines the final classification result. In contrast, in the AmaLgam approach, which was used as a baseline for ABLoTS, the authors empirically set fixed weights for the three suspiciousness scores, which are 0.56 for BLUiR, 0.3 for BugCache and 0.14 for TraceScore. Our DT trained from scratch exhibited the following (non-normalized) feature importance: 0.243 for BLUiR, 0.007 for BugCache, 0.037 for TraceScore, which still does not yield as good results (see Table VII) as the fixed weights determined in AmaLgam.

This discrepancy in feature importance values helped us identify the root cause for the difference in performance results. Rath et al. adopted the implementation of BugCache by Wang et al. [14], where the bug report's fixed date was utilized for the cut-off date, as shown in Fig 4. If one or more bug-fixing commits occurred within 15 days prior to the fixed date, BugCache would recommend the files within these commits (i.e., potentially exactly those files that were changed to fix the bug). However, in a realistic bug localization situation, any file recommendation would only be useful before any of those commits. Thus, for correct evaluation, these commits must not be used.

Figure 4 illustrates such a situation. There is a bug report "HORNETQ-1301" created on 2014-01-09, and fixed on 2014-01-14. Two commits c_6 and c_7 were committed to fix this bug between the created date and fixed date, on 2014-01-09. When BugCache adopts the fixed date as the cut-off date and identifies bug-fixing commits within 15 days, then c_4, c_5, c_6, and c_7 would be taken into consideration and result in a high $Susp^H$ score, according to Eq. 2. Doing so, the DT would learn that the scores by BugCache are very indicative of the actual classification result and hence assign it a high feature importance. However, in practice, c_6 and c_7 are unknown for predicting bug report "HORNETQ-1301", they are foreknowledge about the bug. The right way of implementing BugCache is using the creation date, or any date before the bug's first partial fix implementation. After contacting the authors of both ABLoTS and AmaLgam, AmaLgam's authors stated that they agreed with our finding and that they adopted the wrong date, while authors of ABLoTS stated that they directly reused AmaLgam's implementation.

The incorrectly derived $Susp^H$ scores thus greatly boost the result of the DT. When we utilized BugCache in the correct manner (i.e., use the created date as the cut-off date), DT did not yield results even close to the original performance (even when applying hyperparameter tuning). For comparison, we adopted AmaLgam's composer with a fixed weight for each component: 0.56 for BLUiR, 0.3 for BugCache, and 0.14 for TraceScore. The results of the fixed weight composer are shown in Table VIII, the average MAP, MRR, Top 1. Top 5, and Top 10 are 29.8%, 43.3%, 32%, 56.3% and 64%, respectively. Compared to TraceScore, the results have been improved by 105.8%, 78.7%, 105.2%, 68.4%, and 52.9%, respectively.

Aside from the DT feature importance values, a second discrepancy emerged when we investigated the evaluation data set. In the replication package, the intermediary suspiciousness scores were provided not only as a training set for the DT but also as an evaluation set (i.e., the remaining 20%). When we trained and evaluated with these two data sets, we could replicate the results. However, as outlined above, when obtaining the suspiciousness scores ourselves, we could not. The discrepancy we found was that the evaluation data set contained much fewer evaluation data points (i.e., suspiciousness scores with their classification ground truth) than these projects contained source code files. In other words, for a particular bug, not all source code files were utilized for evaluation but just a subset.[6] Across all projects, the number of candidates ranges from 60 to 70, regardless of actual number of files in the respective project. For the project HORNETQ, for example, even when we select only files for which a TraceScore suspiciousness score and a BLUiR suspiciousness score exist, we obtain around 4500 file candidates. In addition, for some of these files the evaluation data set does not provide any of the three suspiciousness scores at all, just the classification result. Hence, we could not establish how these file candidates have been filtered and why only a subset has been chosen. The paper does not describe this aspect, but rather refers to the evaluation design of Amalgam.

All in all, we found that ABLoTS adopted the wrong cut-off date for BugCache due to having reused the component and configuration from AmaLgam without further investigation, resulting in the incorrect $Susp^H$ scores. Hence, we conclude that ABLoTS performance cannot be replicated.

RQ2.2 Generalizability Since ABLoTS is not replicable, exploring its performance on the extended data set for generalizability evaluation would yield little insight. However, in order to explore how TraceScore would perform when jointly used with the other two components, like in AmaLgam [14], [17], we applied a fixed weight to aggregate the three scores. That is, the suspiciousness score for the source code file s is

[5]Recall that the DT is trained separately for each project.

[6]The identity of the files is not provided in the replication package.

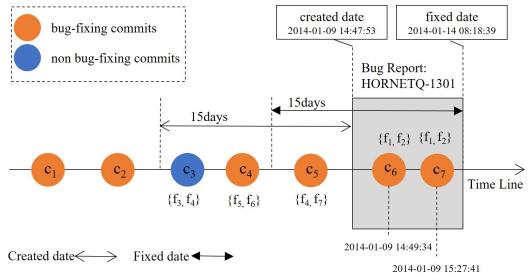

Fig. 4. BugCache using created date vs using fixed date.

TABLE VII
ABLOTS PERFORMANCE ON ORIGINAL DATA SET.

PROJECTS	MAP	MRR	TOP 1	Top 5	Top 10
DERBY	0.076	0.111	0.02	0.171	0.326
DROOLS	0.049	0.06	0.023	0.054	0.097
HORNETQ	0.057	0.067	0	0.056	0.185
IZPACK	0.086	0.11	0.016	0.172	0.375
KEYCLOAK	0.029	0.05	0.006	0.044	0.101
LOG4J2	0.065	0.072	0.011	0.067	0.146
RAILO	0.06	0.077	0	0.1	0.283
SEAM2	0.08	0.105	0.019	0.179	0.333
TEIID	0.056	0.079	0.015	0.104	0.231
WELD	0.02	0.024	0	0.018	0.027
WILDFLY	0.03	0.04	0.007	0.036	0.087
Average	**0.055**	**0.072**	**0.011**	**0.091**	**0.199**

calculated according to Eq. 4, where the value of a and b are set to 0.2 and 0.3 as per prior work.

The results of fixed weight are shown in Table IX. On the additional 16 projects, the fixed weight composer can achieve an average MAP, MRR, Top 1, Top 5, Top 10 as 34.4%, 47.7%, 35.6%, 62.1% and 71.4%, which improves over the single TraceScore by 87.8%, 67.8%, 81.8%, 61.6% and 50.8%, respectively. Compared to the results on the original data set, the average evaluation results over the extended data set are 10% $\sim$ 16% higher (e.g., the average MAP is 34.4 vs 20.2). K-S test ("Fixed Weight" column in Table VI) shows that all the p-values are larger than 5%, so we should reject the hypothesis that the two samples come from different distributions. According to the box plot (right in Fig 3), we can see the distribution on each metric is more concentrated, more similar, and the mean values are closer. Hence, we can conclude that the performance of the fixed weight composer also holds for a larger data set, and we gain confidence in its generalizability.

TABLE VIII
FIXED WEIGHT COMPOSER ON ORIGINAL DATA SET.

PROJECTS	MAP	MRR	TOP 1	Top 5	Top 10
DERBY	0.312	0.478	0.36	0.615	0.725
DROOLS	0.272	0.464	0.339	0.607	0.712
HORNETQ	0.37	0.555	0.426	0.704	0.778
IZPACK	0.37	0.493	0.391	0.594	0.672
KEYCLOAK	0.234	0.377	0.247	0.525	0.595
LOG4J2	0.391	0.541	0.416	0.719	0.753
RAILO	0.286	0.398	0.267	0.567	0.65
SEAM2	0.339	0.402	0.308	0.532	0.583
TEIID	0.12	0.169	0.1	0.208	0.296
WELD	0.252	0.445	0.33	0.562	0.634
WILDFLY	0.334	0.441	0.333	0.565	0.645
Average	**0.298**	**0.433**	**0.320**	**0.563**	**0.640**

$$Susp^{R,S,H}(s) = b * Susp^{H}(s) + (1-b) * (Susp^{R}(s) * a + (1-a) * Susp^{S}(s)) \quad (4)$$

TABLE IX
FIXED WEIGHT COMPOSER ON THE EXTENDED DATA SET.

PROJECTS	MAP	MRR	TOP 1	Top 5	Top 10
ARCHIVA	0.322	0.477	0.347	0.587	0.667
CASSANDRA	0.335	0.462	0.330	0.622	0.741
ERRAI	0.310	0.505	0.389	0.630	0.722
FLINK	0.416	0.560	0.456	0.670	0.752
GROOVY	0.388	0.458	0.331	0.618	0.726
HBASE	0.398	0.528	0.398	0.697	0.778
HIBERNATE	0.234	0.400	0.290	0.551	0.626
HIVE	0.357	0.483	0.343	0.647	0.746
JBOSS-T.-M.	0.370	0.536	0.403	0.701	0.791
KAFKA	0.474	0.623	0.516	0.742	0.844
LUCENE	0.321	0.466	0.336	0.624	0.710
MAVEN	0.337	0.416	0.296	0.546	0.671
RESTEASY	0.257	0.391	0.275	0.536	0.638
SPARK	0.406	0.496	0.379	0.606	0.712
SWITCHYARD	0.160	0.300	0.220	0.407	0.462
ZOOKEEPER	0.422	0.537	0.383	0.745	0.830
AVERAGE	**0.344**	**0.477**	**0.356**	**0.621**	**0.714**

Answering RQ2: *The reported results of ABLoTS are not replicable, because of the incorrect use of the cut-off date in the BugCache component and the sub-optimal configuration of the composer. Consequently, we did not check the generalizability of ABLoTS, since applying an incorrect technique would provide little useful insight. However, with a fixed weight scheme, the results are generalizable on the extended data set.*

C. Discussion

Overall, as shown in Table X, our experimental results suggest that TraceScore is **replicable under *relaxed cut-off date* constraint**, but, **non-replicable under *strict cut-off date* constraint**, where the former can achieve better results. However, in actual applications, the choice between relaxed cut-off date and strict cut-off date is flexible, as commits available at the time when developers perform bug-fixing tasks will be considered for recommendation.

On the extended data set, TraceScore also yields similar results compared with on the original data set, which demonstrates that **TraceScore possesses good generalizability**. However, the results vary more (i.e., some projects exhibit much higher performance, other projects exhibit even lower performance), it is not possible to accurately predict the performance of TraceScore on a new project. Additional investigations are necessary to understand when TraceScore is expected to perform well and under which conditions TraceScore will not yield a lot of benefits.

ABLoTS, in contrast, **is not reproducible** for two main reasons: 1) the authors reused the wrong BugCache implementation from Wang et al. [14] (we confirmed the incorrect use with Wang et al.), which results in the BugCache score greatly boosting the final result; 2) when we adopt the correct BugCache score, we could not duplicate the DT composer because of a lack of details in the original study. We are skeptical whether DT is the right choice for the composer, as also different sampling strategies and hyperparameter tuning yielded a performance worse than the static composer configuration. When we utilized this fixed weight composer proposed by AmaLgam we observed its performance to hold also for the extended data set.

We observed that combining all three scores can improve the TraceScore result by 50% $\sim$ 105% on both data set, which suggests that a combination of different components is likely to outperform any single mechanism. To this end, the choice of composer is a crucial aspect. In preliminary results, that are outside the scope of this paper, we have found that other machine learning and AI techniques can outperform the static composer.

One additional take-away of our replication study is paying attention to the challenge of properly evaluating a technique in the presence of temporal aspects, especially when third-party research outcomes (i.e., BugCache) behave differently than expected. The case of the 15-day interval of BugCache is especially tricky, as for other data sets where commits of a bug predominately happen more than 15 days before the bug's closing date, no such negative side effect would have been noticeable. In the case of ABLoTS sanity checks on the DT's feature importance values would have identified unexpected results (i.e., with BugCache rather than TraceScore dominating the classification result), subsequently triggering confirmation or revision of the composer mechanism.

Overall, the results of this replication study suggest that the state of the art in bug localization is not as useful as prior results have suggested and that further research is still needed to obtain results that are good enough to be useful in practice.

TABLE X
SUMMARY OF RESULTS.

		Replicable	Generalizable
TraceScore	Relaxed cut-off date	Yes	Yes
	Strict cut-off date	No	-
ABLoTS		No	-
Fixed Weight Composer		-	Yes

D. Implications to Future Replication

In this section, we summarize lessons learned through our replication specific to bug localization. The goal is to support researchers in more efficiently and rapidly replicating the approach for a comparative study or as a baseline for novel approaches.

- Data Collection: During the data collection, Rath et al. collected both bug reports and non-bug reports, traceability information between reports, commit logs, commit code change, and constructed links from issues to code change. For the ground truth construction, Rath et al. utilized the modified files and newly added files as the ground truth. However, in our opinion, which files are newly added cannot be predicted by definition. In contrast, removed files are predictable, and should be included in the ground truth. This minor change would not change the technique in the approach, but might impact the evaluation scores.
- Trace Graph Construction: The construction of the trace graph requires previously fixed issues. When replicating, researchers should be careful about the date for artifacts selection. That is, only commits created before the prediction date may be considered.
- BugCache Calculation: For selecting the historical bug-fixing commits, apart from keywords-based selection, also a bug ID can identify bug-fixing commits. More important, as with the trace graph construction, any commit information taken for file candidate scoring must have been already available by the fictive recommendation time (e.g., bug creation date) or at the latest the bug's first partial fix implementation. As we have seen in the replication study, using the bugs fixed date may lead to data leakage.
- Choice of Composer: With a limited set of features (i.e., the three suspiciousness scores), a DT may not be the best choice.

V. THREATS TO VALIDITY

Construct Validity. One possible threat to construct validity is that there is no available open source implementation of ABLoTS approach, which means we have to re-implement it by ourselves. To alleviate this, we carefully read the original study, trying to reproduce it as close as possible. For the BLUiR component, we reused existing open source code from a published paper to reduce possible errors. As for BugCache component, we translate the original implementation from Java into Python with great care. We carefully examined the code and the output to avoid errors.

Internal Validity. From a perspective of internal validity, potential errors can happen in the reproduction (e.g., settings and library usage), which is a common threat to replication studies. We tried out possible settings and compare the results with the original study. Another potential threat is that the open source projects in our data set might have been changed by the day we collected from GitHub. To address this threat, we filter out projects that do not have complete commits information anymore.

External Validity. Regarding external validity, we experimented only on open source Java projects. We encourage future studies to replicate this study with other programming languages as well as commercial projects.

Conclusion Validity. Conclusion Validity could come from the interpretation of the results, which includes the evaluation metrics for evaluation and K-S test for comparison. To mitigate the threat, we adopted the same evaluation metrics adopted

in the original paper. Then the two sample K-S test was utilized to compare the difference of experiment results, as it is sensitive to differences in both location and shape of the empirical cumulative distribution functions of the two samples.

VI. Related Work

Recently, many IRBF approaches have been proposed, which leverage information retrieval techniques to find buggy-prone snippets from all source code candidates. BugLocator calculates similarity between bug reports to recommend similar files to similar bug reports [16]. 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 [51]. Wang et al. combine similar bug reports, code version history, and code structure to find the buggy files [14], [17]. Niu et al. proposed a refactoring-aware traceability model for constructing more accurate code history, which can boost the results of similar bug reports and code version history component [52]. Wen et al. use change logs and change hunks from commit message as alternative of segments of source code files to enable more accurate bug localization [53]. For comparison of state-of-the-art approaches, Lee et al. conducted a generalized and large-scale investigation into six IRBL techniques [24]. Li et al. re-implement six state-of-the-art bug localization approaches and report their effectiveness on 10 Huawei projects [25]. Both studies analyzed the same five state-of-the-art approaches and found lower average results than the original ABLoTS results (e.g., MAP less than 0.4, and MRR less than 0.53). However, neither of these two studies included ABLoTS, which strengthens the usefulness of our study.

VII. Conclusion

In this paper, we conduct a replication study of the ABLoTS approach for bug localization. We recreated the original approach, both on the original data set and an extended data set. We found that the core component of ABLoTS, i.e., TraceScore, is replicable and generalizable under a *relaxed cut-off* constraint, but irreplicable under a *strict cut-off* constraint. ABLoTS is neither replicable nor generalizable because of the adoption of an incorrect cut-off date in the BugCache subcomponent, leading to training data leaking into test data. Also, the chosen technique to combine multiple scores yielded poor results when applied to the correctly derived scores. Our study emphasizes the importance of choosing the proper cut-off dates in evaluating bug localization techniques. As part of future work, we already started investigating alternative information sources and techniques to improve bug localization performance, specifically focusing on techniques to better combine multiple scoring techniques.

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.

References

[1] L. Tan, C. Liu, Z. Li, X. Wang, Y. Zhou, and C. Zhai, "Bug characteristics in open source software," *Empirical software engineering*, vol. 19, no. 6, pp. 1665–1705, 2014.

[2] Ö. Aslan and R. Samet, "Mitigating cyber security attacks by being aware of vulnerabilities and bugs," in *2017 international conference on cyberworlds (cw)*. IEEE, 2017, pp. 222–225.

[3] F. Piessens, "A taxonomy of causes of software vulnerabilities in internet software," in *Supplementary Proceedings of the 13th International Symposium on Software Reliability Engineering*. Citeseer, 2002, pp. 47–52.

[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] 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.

[6] H. Zhang, "An investigation of the relationships between lines of code and defects," in *2009 IEEE international conference on software maintenance*. IEEE, 2009, pp. 274–283.

[7] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *Proceedings of the 28th international conference on Software engineering*, 2006, pp. 361–370.

[8] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Duplicate bug reports considered harmful... really?" in *2008 IEEE International Conference on Software Maintenance*. IEEE, 2008, pp. 337–345.

[9] G. Li, H. Liu, X. Chen, H. S. Gunawi, and S. Lu, "Dfix: automatically fixing timing bugs in distributed systems," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 994–1009.

[10] D. Jeffrey, M. Feng, N. Gupta, and R. Gupta, "Bugfix: A learning-based tool to assist developers in fixing bugs," in *2009 IEEE 17th International Conference on Program Comprehension*. IEEE, 2009, pp. 70–79.

[11] A. Ciborowska and K. Damevski, "Fast changeset-based bug localization with bert," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 946–957.

[12] X. Huo, F. Thung, M. Li, D. Lo, and S.-T. Shi, "Deep transfer bug localization," *IEEE Transactions on software engineering*, vol. 47, no. 7, pp. 1368–1380, 2019.

[13] P. Loyola, K. Gajananan, and F. Satoh, "Bug localization by learning to rank and represent bug inducing changes," in *Proceedings of the 27th ACM International Conference on Information and Knowledge Management*, 2018, pp. 657–665.

[14] 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.

[15] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Bug localization using latent dirichlet allocation," *Information and Software Technology*, vol. 52, no. 9, pp. 972–990, 2010.

[16] 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.

[17] 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.

[18] S. A. Akbar and A. C. Kak, "A large-scale comparative evaluation of ir-based tools for bug localization," in *Proceedings of the 17th international conference on mining software repositories*, 2020, pp. 21–31.

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

[20] C. McMillan, M. Grechanik, D. Poshyvanyk, C. Fu, and Q. Xie, "Exemplar: A source code search engine for finding highly relevant applications," *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 1069–1087, 2011.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] M. Shepperd, N. Ajienka, and S. Counsell, "The role and value of replication in empirical software engineering results," *Information and Software Technology*, vol. 99, pp. 120–132, 2018.

[27] J. C. Carver, "Towards reporting guidelines for experimental replications: A proposal," in *1st international workshop on replication in empirical software engineering*, vol. 1, 2010, pp. 1–4.

[28] G. Haben, S. Habchi, M. Papadakis, M. Cordy, and Y. Le Traon, "A replication study on the usability of code vocabulary in predicting flaky tests," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 219–229.

[29] F. Q. Da Silva, M. Suassuna, A. C. C. França, A. M. Grubb, T. B. Gouveia, C. V. Monteiro, and I. E. dos Santos, "Replication of empirical studies in software engineering research: a systematic mapping study," *Empirical Software Engineering*, vol. 19, no. 3, pp. 501–557, 2014.

[30] O. S. Gómez, N. Juristo, and S. Vegas, "Understanding replication of experiments in software engineering: A classification," *Information and Software Technology*, vol. 56, no. 8, pp. 1033–1048, 2014.

[31] M. Rath and P. Mäder, "The seoss 33 dataset—requirements, bug reports, code history, and trace links for entire projects," *Data in brief*, vol. 25, p. 104005, 2019.

[32] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to information retrieval*. Cambridge university press, 2008.

[33] 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.

[34] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[35] H. Schütze, C. D. Manning, and P. Raghavan, *Introduction to information retrieval*. Cambridge University Press Cambridge, 2008, vol. 39.

[36] E. M. Voorhees *et al.*, "The trec-8 question answering track report." in *Trec*, vol. 99, 1999, pp. 77–82.

[37] M. Rath, D. Lo, and P. Mäder, "Replication data for: Analyzing requirements and traceability information to improve bug localization," 2018.

[38] JIRA, "Jira issue tracking software," https://www.jira.com, 2018.

[39] G. SCM, "Git scm," https://www.git-scm.com, 2018.

[40] A. Bachmann and A. Bernstein, "Software process data quality and characteristics: a historical view on open and closed source projects," in *Proceedings of the joint international and annual ERCIM workshops on Principles of software evolution (IWPSE) and software evolution (Evol) workshops*, 2009, pp. 119–128.

[41] X. Ye, R. Bunescu, and C. Liu, "Learning to rank relevant files for bug reports using domain knowledge," in *Proceedings of the 22nd ACM SIGSOFT international symposium on foundations of software engineering*, 2014, pp. 689–699.

[42] R. Baeza-Yates, B. Ribeiro-Neto *et al.*, *Modern information retrieval*. ACM press New York, 1999, vol. 463.

[43] NLTK, "Nltk library," http://www.nltk.org, 2022.

[44] K. S. Jones, "A statistical interpretation of term specificity and its application in retrieval," *Journal of documentation*, 1972.

[45] scikit learn, "scikit-learn," https://scikit-learn.org/stable/, 2022.

[46] 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.

[47] 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.

[48] T. Strohman, D. Metzler, H. Turtle, and W. B. Croft, "Indri: A language model-based search engine for complex queries," in *Proceedings of the international conference on intelligent analysis*, vol. 2, no. 6. Citeseer, 2005, pp. 2–6.

[49] Imblearn, "Imblearn," https://imbalanced-learn.org/stable/, 2022.

[50] F. J. Massey Jr, "The kolmogorov-smirnov test for goodness of fit," *Journal of the American statistical Association*, vol. 46, no. 253, pp. 68–78, 1951.

[51] 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.

[52] F. Niu, W. K. G. Assunçao, L. Huang, C. Mayr-Dorn, J. Ge, B. Luo, and A. Egyed, "Rat: A refactoring-aware traceability model for bug localization," in *2023 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, accepted.

[53] 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.