UniLoc: Unified Fault Localization of Continuous Integration Failures

FOYZUL HASSAN, University of Michigan-Dearborn, USA
NA MENG, Virginia Tech, USA
XIAOYIN WANG, University of Texas at San Antonio, USA

Continuous Integration (CI) practices encourage developers to frequently integrate code into a shared repository. Each integration is validated by automatic build and testing such that errors are revealed as early as possible. When CI failures or integration errors are reported, existing techniques are insufficient to automatically locate the root causes for two reasons. First, a CI failure may be triggered by faults in source code *and/or* build scripts, while current approaches consider only source code. Second, a tentative integration can fail because of build failures *and/or* test failures, while existing tools focus on test failures only. This paper presents UniLoc, the first unified technique to localize faults in *both* source code *and* build scripts given a CI failure log, without assuming the failure's location (source code or build scripts) and nature (a test failure or not). Adopting the information retrieval (IR) strategy, UniLoc locates buggy files by treating source code and build scripts as documents to search and by considering build logs as search queries. However, instead of naïvely applying an off-the-shelf IR technique to these software artifacts, for more accurate fault localization, UniLoc applies various domain-specific heuristics to optimize the search queries, search space, and ranking formulas. To evaluate UniLoc, we gathered 700 CI failure fixes in 72 open-source projects that are built with Gradle. UniLoc could effectively locate bugs with the average MRR (Mean Reciprocal Rank) value as 0.49, MAP (Mean Average Precision) value as 0.36, and NDCG (Normalized Discounted Cumulative Gain) value as 0.54. UniLoc outperformed the state-of-the-art IR-based tool BLUiR and Locus. UniLoc has the potential to help developers diagnose root causes for CI failures more accurately and efficiently.

CCS Concepts: • **Software and its engineering** → Software maintenance tools;

Additional Key Words and Phrases: Fault localization, CI failures, information retrieval (IR)

1 INTRODUCTION

As an emerging software engineering practice [18], Continuous Integration (CI) [24] enables developers to identify integration errors in earlier phases of the software process, significantly reducing project risk and development cost. Meanwhile, *CI poses higher demands for efficient fault localization and program repair techniques to improve the continuous success of the practice*. Specifically, prior work reports that, on average, the Google code repository receives over 5,500 code commits per day, which makes the Google CI system run over 100 million test cases [65]. When any commit is buggy, the corresponding and follow-up integration trials ("**CI trials**" for short) will keep failing until the bug is fixed by another commit. A long-standing CI failure can stop developers from testing commits effectively [6] and diminish people's confidence in adopting CI [7]. Existing fault localization (FL) techniques either rely on bug reports or test failures to locate bugs in source code [21, 33, 79, 84]. However, CI failures bring new challenges to these techniques.

Authors' addresses: Foyzul Hassan, University of Michigan-Dearborn, USA, foyzul@umich.edu; Na Meng, Virginia Tech, USA, nm8247@cs.vt.edu; Xiaoyin Wang, University of Texas at San Antonio, USA, xiaoyin.wang@utsa.edu.

1049-331X/2023/5-ART $15.00
https://doi.org/10.1145/3593799

Challenge 1: Faulty build scripts. Unlike traditional fault localization scenarios where only source code is assumed to be buggy, CI failures can also be triggered by build configuration errors and environment changes. In other words, build scripts can be faulty, but current techniques do not examine these scripts. A recent study [56] on 1,187 CI failures shows that 11% of failure fixes contain build script revisions only, and 26% of failure fixes involve revisions to both build scripts and source files. The study indicates that without considering build scripts, existing techniques are incapable of handling a large portion of CI failures (i.e., 37%).

Challenge 2: Non-Test Failure. A tentative integration may not proceed smoothly due to failures other than test failures, while existing fault localization techniques mainly consider test failures to locate bugs. Specifically, Rausch et al. [56] found that among the five major reasons for CI failures, dependency resolution, compilation, and configuration errors account for 22% of the scenarios, while quality checking errors and test failures separately take up 13% and 65%. This finding implies that current tools are applicable to at most 65% of CI failures. To facilitate discussion, we name all failures other than test failures as ***Non-Test Failure***. Existing approaches are unable to localize bugs for non-test failures.

To overcome the above challenges, we developed a novel approach—UniLoc—to suggest a ranked list of candidate buggy files given a CI failure log. Unlike existing fault localization techniques, UniLoc takes both source files and build scripts into consideration, and conducts unified fault localization to diagnose both test failures and non-test failures. The key insight behind UniLoc is that the CI failure log, the source files and build scripts, and file changes in commit history can complement and cross validate with each other to reduce additional noises in this heterogeneous environment. In particular, we adopted the information retrieval (IR) strategy by treating files as documents (D), and by considering the failed logs (L) as search queries to retrieve documents. Similar to prior work [58, 73], UniLoc also extracts Abstract Syntax Trees (ASTs) from source files and build files to divide large documents into smaller ones. However, existing IR-based tools cannot perform unified fault localization for two reasons:

1) **Noisy data in *L*.** Prior IR-based fault localization (IRFL) work uses a given bug report as a whole to retrieve related source code, assuming that everything mentioned in the report is relevant. However, CI logs are usually lengthy and contain lots of information irrelevant to any failure. Existing approaches do not refine L to reduce the noise.
2) **Noisy data in *D*.** Prior IRFL work relies on textual relevance to locate bugs given a bug report. However, textually relevant files may not be involved in a failed CI trial, depending on the build configuration. Existing tools do not refine D based on the build-target dependencies between modules.

UniLoc solves the above-mentioned issues by (1) optimizing queries to remove noisy information, (2) optimizing documents to remove files unrelated to a failing build, and (3) tuning candidate ranking to prioritize the most recently changed files. To optimize queries, the work applies a text diff algorithm between the passed build log and failed build log to extract failure related text. Even after acquiring the failure-related texts, the query can still have noises like time and build-process-related information. Such repeated noises are removed through a similarity based approach. At the same time, search space is optimized by extracting important source code and build scripts applying AST analysis to them. Such AST analysis for search space optimization includes only important terms of software entities (e.g., class names, methods names, build dependency names, etc.) instead of considering full source code and build scripts that may contain noisy terms. Details of AST-Based entity extraction is discussed in subsubsection 3.2.2. Moreover, static build dependency analysis was applied to rule out files and project modules that are not associated with the CI failure. Finally, candidate ranking was optimized based on the heuristics that the recent changes are more likely to be the root cause of the CI failure.

To evaluate UniLoc, we collected 700 real CI failure fixes in 72 GitHub projects from the TravisTorrent dataset [19]. We used earlier 100 fixes for parameter tuning and the remaining 600 fixes for evaluation.

As with prior work [21, 51, 58, 84], we evaluated UniLoc's effectiveness by measuring Top-N (Recall at Top N), MRR (Mean Reciprocal Rank), MAP (Mean Average Precision), and NDCG (Normalized Discounted Cumulative Gain). Our evaluation shows that UniLoc located buggy files with 65% Top-10, which means that among 65% of the scenarios, UniLoc successfully included buggy files in the Top-10 recommendations. On average, the MRR, MAP and NDCG values of UniLoc were 0.49, 0.36 and 0.54, respectively.

This paper is the first work on unified fault localization of both test failures and non-test failures. To compare UniLoc with existing code-oriented IRFL, we applied widely used IRFL approaches BLUiR [58] and Locus [73] to the 600 bug fixes. [22, 38]. The MRR, MAP and NDCG values of BLUiR were 0.29,0.19 and 0.39 respectively, much lower than those values of UniLoc. For the case of Locus, MRR, MAP and NDCG values are 0.12, 0.09 and 0.22, respectively, which are also much lower than those values of UniLoc. Such results suggest that existing IR-based FL techniques can localize CI failures to some extent, but they may not effectively localize CI failures in many cases and their overall performance is lower than UniLoc. Furthermore, UniLoc optimizes (a) queries, (b) the search space, and (c) candidate ranking to improve fault localization. To learn how sensitive UniLoc is to each applied optimization strategy, we evaluated three variants of UniLoc with one strategy removed for each variant. Our experiment shows that all three strategies are useful, and the optimization of candidate ranking boosts the effectiveness most significantly.

We summarize the contributions of this paper as follows:

- We developed a unified fault localization approach UniLoc that considers both source code and build scripts to diagnose CI failures. UniLoc includes novel techniques to extract optimized queries from failed build logs, to generate optimized document sets from source files and build scripts, and to rank suspicious files with IR scores and commit history data.
- We constructed a data set of 700 CI failure fixes together with the related failure-inducing commits from real-world projects in Github. We open-sourced the data and implementation to facilitate future research in CI failure repair. Our data and program are separately available at https://sites.google.com/view/uniloc and https://github.com/foyzulhassan/UniLoc.
- We conducted a comprehensive evaluation to evaluate the effectiveness of UniLoc. We explored how UniLoc works differently from source-code-oriented IR-based fault localization techniques. We also investigated how different optimization strategies affect the effectiveness of UniLoc.

The organization of the paper is as follows. After describing the background of this work in Section 2, we introduce UniLoc in Section 3. Section 4 explains evaluation and Section 6 discusses the generalization of our approach. We expound on the related works and conclusion in Section 7 and Section 8, respectively.

2 BACKGROUND

This section first clarifies terminology (Section 2.1), and then introduces the IR technique we used (Section 2.2). Finally, it explains the project dependencies manifested by build scripts (Section 2.3).

2.1 Terminology

This paper uses the following terms:

CI trial is the integration process of validating a commit with an automated build and automated tests.

CI log is the log file generated for a CI trial to record any intermediate status as well as the outcome—"*passed*" or "*failed*".

Passed log is the log file generated for a successful CI trial.

Failed log is the log file produced for a failed CI trial.

CI failure fix is a program commit whose application updates the CI trial outcome from "failed" to "passed".

CI failing-inducing commit is a commit that produces a failed CI trial before a CI failure fix is applied.

```
include "spock-bom"
include "spock-core"
include "spock-specs"
include "spock-spring"
include "spock-spring:spring2-
     test"
...
```

(a) Script settings.gradle of the whole project

```
...
dependencies {
compile project(":spock-core")
 ...
}
...
```

(b) Script report.gradle of the spock-report sub-project

```
...
dependencies {
  testCompile project(":spock-core
       ")
...
}
...
```

(c) Script specs.gradle of the spock-specs sub-project

Fig. 1. Three *.gradle files used in *spockframework/spock* declare sub-projects and specify dependencies between the projects

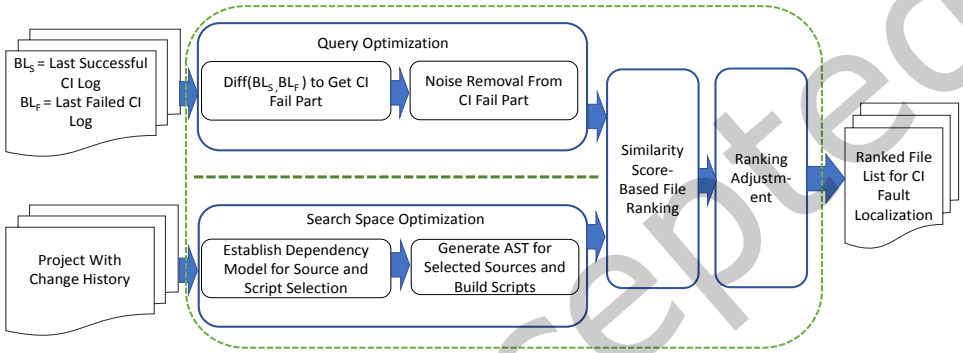

Fig. 2. UniLoc Architecture

2.2 Information Retrieval (IR)

Given a text query q, an Information Retrieval system searches among the document corpus (D) for relevant documents. To retrieve documents relevant to q, the IR system computes a similarity score between each document $d \in D$ and q, and ranks documents in descending order of the scores. Below is a frequently used formula for similarity calculation:

$$Sim(q,d) = cos(q,d) = \frac{\overrightarrow{V_q}.\overrightarrow{V_d}}{|\overrightarrow{V_q}||\overrightarrow{V_d}|} \quad (1)$$

$\overrightarrow{V_q}$ and $\overrightarrow{V_d}$ are term weight vectors of q and d, while $Sim(q,d)$ is the cosine similarity of the two vectors. In particular, the term weight values in each vector are determined by term frequency (TF) and inverse document frequency (IDF). In a typical IRFL approach, source files are treated as documents, while bug reports are used as queries. Similarity scores are computed to assess how probably a file is buggy.

2.3 Project Dependencies

A build system is an infrastructure to convert source code into artifacts such as modules, libraries, and executable binaries [47]. A build script specifies how to generate and test artifacts for software projects. In Gradle, a big project can be divided into several sub-projects. The dependencies between sub-projects are defined in build scripts, where the overall project is referred to as *root*. When developers commit program changes, not every sub-project needs to be rebuilt or retested. Instead, the build system only compiles and tests the sub-projects being changed and those sub-projects depending on the changed ones.

Figure 1 presents three Gradle scripts defined in the project *spockframework/spock*. In Figure 1 (a), script *settings.gradle* shows that the project includes multiple sub-projects, such as *spock-bom* and *spock-core*. In Figure 1 (b), `compile/testcompile project(":spock-core")` means that *spock-core* is needed for Gradle to compile the owner sub-project. Figure 1 (c) indicates that *spock-core* is needed to compile and test *spock-specs*.

The dependencies between sub-projects can be utilized in UniLoc. For instance, when *spock-report* does not compile, only its source code and those sub-projects or libraries on which *spock-report* transitively depends should be examined.

3 APPROACH

Figure 2 overviews the design of UniLoc. We envision UniLoc to be used by developers when they notice a failed log ever since the most recent passed log.

Specifically, UniLoc takes in three inputs: the most recent passed build log BL_s, the failed build log BL_f immediately after BL_s, and the commit history H which includes the most recent passed version V_s, the failed version V_f immediately after V_s, and the failure-inducing single commit (file changes) C_f between the two versions V_s and V_f. Prior studies on quality issues of CI process [67, 82] suggested that developers should follow the process of immediate small integration rather than waiting for large or merged integration to avoid spaghetti of error during integration. Therefore, we consider that developers are integrating code immediately and V_f is the commit that introduced the build failure. To suggest a ranked list of potential buggy files, UniLoc consists of the following four phases:

- Phase 1 compares BL_f with BL_s to locate failure-relevant description FD and to compose a query q (Section 3.1).
- Phase 2 retrieves V_f from H and creates project dependency graphs based on build scripts. With the FD from Phase 1 and recognized dependencies, this phase refines the search scope (Section 3.2).
- Phase 3 compares each document within the scope against q to calculate the similarity score and rank documents accordingly (Section 3.3).
- Phase 4 retrieves C_f, the failure-inducing commit, and extracts the file-level change information to improve the ranking formula (Section 3.4).

3.1 Query Optimization

To query for buggy documents with BL_f, we decided not to use every word in the log. This is because although there can be thousands of lines of build information in a failed log, only a very small portion of those lines are failure-related. Including unrelated information in a query will cause severe noises when we match q with documents. We extracted the failure-related part from BL_f by taking two steps: (i) query optimization with text diff (Section 3.1.1), and (ii) noise removal with text similarity (Section 3.1.2).

3.1.1 Query Optimization with Text Diff. We observed that a failed log could contain duplicated description with a passed log. Such duplicated fragments are usually less informative than those fragments unique to the failed log. Inspired by prior work that uses binary file differentiation to locate unreproducible builds [57], we applied a textual differentiation algorithm—Myers [52]—to BL_f and BL_s to identify any failure-related part in BL_f. Myers finds the longest common sub-sequence of two given strings. The algorithm is based on the concept of finding the shortest edit script that can be modeled as a graph search. Myers algorithm is used as a popular diff utility as a comparison tool that displays line-by-line deletions and insertions for transforming one file into another. Example 1 shows a passed log (commit:0545247) and a failed log (commit:bf25fdf) of the project *BuildCraft/BuildCraft*. BL_f and BL_s denote the CI failed log at version V_f and the latest successful CI log at version V_s before version V_f. The delta between V_s and V_f can include source files, or build files or both. We only considered the latest successful version before V_f, as other successful versions may include logs generated by

Example 1 CI Log Diff *(BuildCraft/BuildCraft)*

```
CI Log Part for Commit ID: 0545247
...
Resolving deltas: 13% (69/522)
Resolving deltas: 16% (84/522)
...
Download http://repo1.maven.org/maven2/com/google/collections/google-collections/1.0/google-collections-1.0.jar
:checkstyleMain
...
BUILD SUCCESSFUL
...
Done. Your build exited with 0.
_____________________________________________
CI Log Part for Commit ID: bf25fdf
...
Resolving deltas: 71% (377/524)
Resolving deltas: 72% (379/524)
...
Download http://repo1.maven.org/maven2/com/google/collections/google-collections/1.0/google-collections-1.0.jar
:checkstyleMain[ant:checkstyle]
     /BuildCraft/BuildCraft/common/buildcraft/core/statements/ActionMachineControl.java:16: Wrong order for
     'cpw.mods.fml.relauncher.Side' import.
[ant:checkstyle]  /BuildCraft/BuildCraft/common/buildcraft/core/statements/StatementParameterDirection.java:16:8:
     Unused import - buildcraft.api.core.NetworkData.
...
FAILED
FAILURE: Build failed with an exception.
```

successfully integrating code segments or build logic that is not part of the failure description and may create additional noise. In Example 1, after utilizing Myers algorithm, it shows the unique fragments in the passed log are highlighted with gray, while the unique fragments in the failed log are highlighted with red and yellow. UniLoc can extract these fragments using Myers.

3.1.2 Noise Removal with Text Similarity. With the above-mentioned text diff, we can divide BL_f into two parts: $PART_s$—the part that successfully matches certain segment(s) in BL_s, and $PART_f$—the unique part of BL_f. Actually, $PART_f$ may still contain segments unrelated to the failure. This is because some program logic changes (e.g., adding new tests), environment changes (e.g., removing dependencies on libraries), and random issues (e.g., multithreading) can also make BL_f look different from previous logs. In Example 1, yellow-colored text block presents such an example of noise due to the change in the download process.

Failure-irrelevant lines in $PART_f$ are not responsible for CI failures, and they may be similar to fragments in $PART_s$. To further remove such failure-irrelevant noise, we conducted line-to-line comparison between $PART_f$ and $PART_s$ using Myers algorithm [17]. If the similarity between any two lines (l_f, l_s) is above a threshold lt, we consider the lines to match, so l_f should be removed from $PART_f$. Since error-related segments of $PART_f$ are usually very different from the normal output of $PART_s$, this noise removal approach is unlikely to remove error-related segments. With this step, we can remove the yellow segment in Example 1 while retaining the two checkstyle errors.

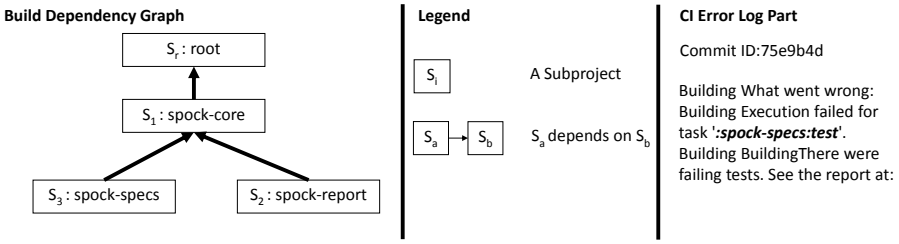

Fig. 3. Sub-Project Dependency Graph

To decide the optimum value of lt, we used 100 CI failure fixes in our dataset (see Section 4.1) as the tuning set. Note that the 100 CI failures are not used in the evaluation set and they are all chronologically earlier than the CI failures in the evaluation set. We found 0.9 to be optimal and thus set $lt = 0.9$ by default. Finally, we denote the refined failure-relevant part with $PART'_f$, which is used by UniLoc to compose a query q.

3.2 Search Space Optimization

In addition to optimizing queries, we also optimized the search space for better fault localization. This phase consists of two steps: i) dependency-based sub-project selection, and ii) AST-based entity name extraction.

3.2.1 Dependency-Based Sub-Project Selection. As described in Section 2.3, Gradle build scripts specify dependencies between sub-projects. We developed a parser to analyze those build scripts and to extract the dependencies. With the extracted dependencies for each software project, we constructed a build graph $B = (S, E)$, where $S = \{s_1, \ldots, s_p\}$ is the set of sub-projects and E is the set of dependency edges. There is a directed edge from s_i to s_j if and only if s_i depends on s_j. Gradle builds each sub-project s_i only after building all the projects on which s_i depends.

To reduce search space, UniLoc finds the mentioned sub-project s_l that is closest to the CI failure in $PART'_f$. Starting from s_l, UniLoc traverses the dependency graph to find all sub-projects on which s_l depends. UniLoc then includes the source files and build scripts of these sub-projects into the search scope, because only these documents are involved in the CI trial for s_l and may be responsible for the failure.

Figure 3 shows part of the dependency graph of *spockframework/spock*. According to the graph, *spock-core* depends on *root* while being dependent on by *spock-specs* and *spock-report*. In this Figure, the CI Error Log Part shows that the failure occurs in *spock-specs*. With the project dependencies, UniLoc can skip *spock-report* when searching for buggy files because this project has no dependency relationship with *spock-specs*.

3.2.2 AST-Based Entity Extraction. Prior work [58] shows that IRFL techniques work better, if the search space includes only names of software entities (e.g., class names, methods names) instead of all source code that contains noisy details. Although we could adopt this technique [58] for source code, the proper software entities to be used may be different in the FL scenarios of CI failures and there is no counterpart for build scripts. Therefore, we developed a unified mechanism on source files and build scripts to identify the top software entities to be included in the search space.

First, we used UniLoc to generate ASTs for sources files / Gradle build scripts, and to extract AST nodes and their textual values. Then, for each subject project in our tuning set, we searched for the textual values in its build scripts and build failure logs. Finally, we counted the frequency when the value of a specific type of AST node can be found in build failure logs. We consider four (we use four to be consistent with [58]) AST node types with the highest frequency as top software entities. Note that this is the same tuning set we used for query noise removal (See Section 3.1.2). In Tables 1 and 2, we show the statistical results for Java source code and Gradle scripts. The second column shows the number of build failures where at least one AST node of the type has its

Table 1. Top Software Entities Source Code

AST Node	Frequency	Example
Declaration names	100	private long **pID**;
Method names	99	public ISlot **create**(...)
Class names	84	public class **AddSlotFactory**
Import items	62	import **org.spockframework.runtime**;

Table 2. Top Software Entities in Build Scripts

AST Node	Frequency	Example
Dependency items	100	dependencies { compile(**'com.google.guava: guava:13.0.1'**)}
Property definitions	100	**classifier = 'standalone'**
Module names	82	project(**':moco-core'**)
Task definitions	43	task **sourcesJar(type: Jar)**

textual values appearing in the failed build log. The bold part of the third column shows an exemplar textual value of the AST node type (the rest part of the column gives some context). As shown in Tables 1 and 2, *field names*, *method names*, *class names*, and *import items* are top four software entities in Java source code; *dependency items*, *property definitions*, *module names*, and *task definitions* are top four software entities in Grade build scripts. Therefore, we included only the textual values of these entities in the search space.

3.3 Similarity Score-Based File Ranking

Traditionally, there are two main types of automatic FL techniques: spectrum-based vs. IR-based. Spectrum-based techniques exploit the execution coverage information of passed and failed tests to rank suspicious files. The reason why we chose to take an IR-based approach is two-fold. First, faults in a project may exist in source files of various programming languages and build scripts. Spectrum-based FL techniques must instrument different language implementations simultaneously to profile executions and analyze test failures. However, we intended to locate faults in build scripts even though no test failure exists.

Second, the instrumentation by spectrum-based techniques can modify program behaviors, introduce runtime overhead to program executions, and can make some failures impossible to reproduce (e.g., flaky tests). Without reproducing those failures, spectrum-based techniques cannot locate any bug.

We reused the implementation in Lucene [3] for the IR technique described in Section 2.2. Given $PART'_f$ and the refined search scope, this implementation creates a query vector q and a set of document vectors D. In each document $d \in D$ (i.e., Java file or build script), we conducted a separate search for each type of software entities. For Java files, we calculated the similarity scores between q and d for the set of entity types includes: *field names*, *method names*, *class names*, and *import items*. Then we leveraged the average value between the similarity scores to compute the overall similarity between q and d:

$$Score(q, d) = \frac{1}{|EntityTypes|} \sum_{et \in EntityTypes} Sim(q, d_{et}) \quad (2)$$

In Formula (2), if d is a build script, the set of entity types includes: *dependency item*, *property definition*, *module name*, and *task definition*. The score is within [0, 1]. The higher score a document has, the higher it is ranked.

3.4 Ranking Adjustment

To further improve file ranking, we leveraged an intuition that the changed files in a failure-inducing commit and the file names mentioned in $PART'_f$ are more suspicious than other files. If developers noticed a failed version V_f after the most recent passed version V_p, we considered all changed files between V_f and V_p together with the files mentioned in $PART'_f$ to be suspicious files. We considered only changes between V_f and V_p because prior study [29] identified that if there is CI failure, in most cases, CI outcome remains unchanged with a median of four CI failures and a maximum of 760 consecutive CI failures. Among these consecutive CI failures, identifying the culprit commit is challenging as failures can be just failures related to dependency of the first failure, or they could separate independent failures due to the concurrent nature of the code commit. Additionally, applying a binary search-based approach to detect culprit commit might not work in cases where commits are dependent on each other and may take a long processing time if there are multiple consecutive CI failures. For simplicity, we assume that after each CI failure, developers will analyze and debug the failure, so we only considered V_f after the most recently passed version V_p. Furthermore, we defined the following formula to adjust similarity scores for documents:

$$FinalScore(x) = \begin{cases} Score(q,d)^{\alpha}, & \text{if } d \text{ is a suspicion file} \\ Score(q,d), & \text{otherwise} \end{cases} \tag{3}$$

In Equation 3, if a file d is suspicious, we raise the score to $Score(q,d)^{\alpha} (0 \leq \alpha \leq 1)$; otherwise, the score $Score(q,d)$ remains the same. Note that in Xuan et al.'s prior work [80], they defined a formula to boost suspicious scores when certain files were recently changed, and we were inspired by their formula. We opted to do file-level fault localization in file level as most of CI failures are required to fix in multiple files and multiple lines, which is ill-suited for line-level fault localization [44].

To find the optimal value of α, we varied α from 0.0 to 1.0 with 0.1 increment, and conducted experiments with the parameter-tuning dataset mentioned in Section 3.1. The experiments showed that 0.1 is the best setting, so we set $\alpha = 0.1$ by default.

4 EXPERIMENTS AND ANALYSIS

In this section, we will first introduce our dataset (Section 4.1) and evaluation metrics (Section 4.2). We will then describe the research questions we explored (Section 4.3), and finally discuss the evaluation results (Section 4.4).

4.1 Dataset

We constructed our evaluation dataset based on TravisTorrent [19], which is a dataset of CI builds. We used the SQL dump version of the dataset dated February 8, 2017, which version contains the build data collected from 2011-08-29 to 2016-08-30. For Java projects, TravisTorrent provides CI log data for three build systems: Ant, Maven and Gradle, and each CI log is a plain text file. A recent study [62] shows that more than 50% of top GitHub projects are using Gradle as their build configuration tool, so we focused our approach implementation and evaluation on Gradle-based projects.

In TravisTorrent, we first extracted all CI failures for any Java project built with Gradle. For each CI failure, we further extracted the most recently passed version as V_s, and its corresponding log as BL_s. We also extracted the failed version immediately following V_f and its corresponding log as BL_f. These are the information available in the FL scenario of each build failure, so we used them as the input of our tool. We further extracted the following failure-to-pass transition to find out how the failure is fixed. In particular, all changed files in the commit that leads to the first following passed build are considered ground truth of fault localization for this build failure. Here we follow prior works in FL [37, 55, 69] to use all changes in the fixing commit as the ground truth. In CI,

developers only change code when the build failure is confirmed to be related to a defect. With the restriction to have a failure-to-pass transition commit, we actually ruled out the flakiness-related failures that are reported in recent CI research work [25, 26].

With our data collection method, we identified 700 CI failures fixes from 72 Gradle-based projects. As shown in Table 3, we used the chronologically earliest 100 of the fixes for parameter tuning (see Section 3.1 and Section 3.4), and for software entity selection in Java source code and Gradle build scripts (see Section 3.2.2). We used the remaining 600 fixes to evaluate the effectiveness of our fault localization techniques. In the dataset, the average number of source files and the average number of build script files per project is 444.32 and 9.24, respectively. Even though the number of build scripts per project is much lower than the source file, the localization of build-related CI failures is challenging due to the abstraction of build logic and limited domain knowledge about the build process among the developers. Even in many cases, CI logs suggest that the failure is in the source file, but in reality, the fault is in the build script(see Example 4). Furthermore, there is no or minimal support for debugging build scripts, which complicates the build script fault localization process. For example, Example 2 shows a CI failure where the log suggests that the failure is related to a plugin and it's in the Hystrix module. However, the fix shows that the failure is due to `io.reactivex:rxjava` dependency and it's in the hystrix-core module build script. Such an example shows that fixing CI failures is challenging due to the high-level abstraction of build scripts and requires deep domain knowledge of build logic and project structure. Moreover, existing debugging tools can't be utilized to analyze such CI failures. So, having a tool even for file-level FL can substantially reduce developer effort to localize build script related CI failures.

Example 2 A Build Failure Fix that Requires CI specific Knowledge *(Netflix/Hystrix: Build Fix Version:bc86bdb)*

```
 * Where:
 Build file '~/Hystrix/build.gradle' line: 11
 * What went wrong:
 An exception occurred applying plugin request [id: 'nebula.netflixoss', version: '3.2.3']
 > Failed to apply plugin [class 'nebula.plugin.bintray.NebulaBintrayPublishingPlugin']
    > Walk failure.
```

```
------------------------------------------------
File: /hystrix-core/build.gradle
   dependencies \{
    compile 'com.netflix.archaius:archaius-core:0.4.1'
    compile 'io.reactivex:rxjava:1.1.0'
    compile 'io.reactivex:rxjava:1.1.1'
    compile 'org.slf4j:slf4j-api:1.7.0'
```

More importantly, we manually inspected the 700 failed logs and their corresponding fixes. For manual inspection, first we classified failures into test failures and non-test failures using the build log analysis mentioned in prior work [30]. Then the first author manually confirmed the classification by inspecting the logs of each failure. We clustered the data based on (1) the failure type and (2) bug locations. As shown in Table 4, 316 (45%) CI failures are test failure and 384 (55%) failures are due to non-test failure. This observation implies that existing FL techniques cannot handle most CI failures in our data set because they mainly rely on the existence of test failures. Moreover, among 316 test failure, 54 failures (17%) require fixes in build scripts. Although the Spectrum-Based fault localization (SBFL) technique works more precisely for test failures, current SBFL techniques only focus on source files without handling build scripts. Furthermore, since SBFL techniques rely on instrumentation, it would be difficult to adapt them for build scripts due to the variety of build tools and plug-ins. In comparison,

Table 3. Dataset Summary

Type	Count
Total Number of Projects	72
Average Number of Source Files Per Project	444.32
Average Number of Build Files Per Project	9.24
Average LOC for Source Files	129.89
Average LOC for Build Files	162.69
Maximum Number of Fix From Single Project	119
Minimum Number of Fix From Single Project	1
Average Number of Fix Per Project	9.72
Total Number of Fix	700
Average Number of Buggy Files Per CI Failure	6.34
Average Number of Buggy Source Files Per CI Failure	5.94
Average Number of Buggy Build Scripts Per CI Failure	0.40
Tuning Set Size	100
Evaluation Set Size	600

Table 4. Failure Types and Bug Locations

	Total	Only Source Fix	Only Build Script Fix	Both File Type Fix
Test Failure	316	262	22	32
Non-Test Failure	384	244	73	67

our new approach is more generic and more applicable because it can locate bugs in both code and scripts, no matter whether the failures are related to tests or not.

Additionally, 95 CI failure fixes (14%) changed both source files and build scripts, while another 99 failure fixes (14%) changed build scripts only. It implies that when current FL tools do not analyze build scripts to locate bugs, they can miss bug locations for many CI failures. In particular, Example 3 shows a CI failure related to the usage of a tool Crashlytics. To fix the failure, both a build script and a Java file were modified. In the build script, `enableCrashlytics` was set to false. In source code, the import declaration of class `com.crashlytics.android.Crashlytics` was removed. Example 4 shows another CI failure, which is triggered by a test failure. In the example, the unit test throws an exception `ClassNotFoundException` because of a missing dependency. Consequently, the related fix added the project dependency to a build script. To fix build script related failures like `enableCrashlytics`, developers need specialized knowledge and may need to spend a long time. In our dataset, source-code-only fixes account for 506 of 700 failures, and for the rest of 194 failures, their fixes involve at least one build script. Since in CI pipeline, developers need to fix CI failures as soon as possible to allow further integration, the commit time between failed build and successful build is a good indication of time spent for CI failures. According to our analysis, for source-only related fixes, the median time spent is 43.5 minutes and for build script related 194 failures, the median time spent is 73 minutes. This analysis also indicates the complexity of build script related failures.

Our prior finding indicates that a considerable portion of CI failures are not triggered by test failures or fixed by modifications in source code. **Our dataset also demonstrates the need to develop a general fault localization technique that (1) analyzes both source files and build scripts, and (2) handles non-test failure in addition to test failures.**

4.2 Evaluation Metrics

We used the following four widely used metrics [21, 51, 55, 77, 81, 84] to measure the effectiveness of FL techniques.

Example 3 A Build Failure Fix with Both Build Script and Source Code Change *(abarisain/dmix: Build Failure Version:2007058, Build Fix Version:86a0af2)*

```
* What went wrong:
Execution failed for task ':MPDroid:crashlyticsCleanupResourcesFossDebug'.
&gt; Crashlytics Developer Tools error.

------------------------------------------------
File:MPDroid/build.gradle
    foss {
       versionName defaultConfig.versionName + "-f"
       +ext.enableCrashlytics = false
     }

------------------------------------------------
File:~/mpdroid/MPDApplication.java
-import com.crashlytics.android.Crashlytics;
```

Example 4 A Test Failure Having Fix in Build Script *(jphp-compiler/jphp: Build Failure Version: a148d3c, Build Fix Version: 1608e0c)*

```
1 warningorg.develnext.jphp.json.classes.JsonProcessorTest &gt; testBasic FAILED
       Caused by: java.lang.ClassNotFoundException at JsonProcessorTest.java:21
1 test completed, 1 failed
:jphp-json-ext:test FAILED
FAILURE: Build failed with an exception.

------------------------------------------------
       testCompile 'junit:junit:4.+'
    + testCompile project(':jphp-zend-ext')
    testCompile project(':jphp-core').sourceSets.test.output
```

- **Recall at Top N (Top-N)** calculates the percentage of CI failures, which have at least one buggy file reported in the top N (N=1,5,10, ...) ranked results. Intuitively, the more failures have their buggy files recalled in Top-N results, the better an FL technique works.
- **Mean Reciprocal Rank (MRR)** measures the precision of FL techniques. Given a set of queries, MRR calculates the mean of Reciprocal Rank values for all queries. The higher the value, the better. The Reciprocal Rank (RR) value of a single query is defined as:

$$RR = \frac{1}{rank_{best}} \tag{4}$$

  Specifically, $rank_{best}$ is the rank of the first correctly reported buggy file. For example, for a given query, if 5 documents are retrieved, and the 3^{rd} and 5^{th} are relevant, then RR is $\frac{1}{3} = 0.33$.
- **Mean Average Precision (MAP)** measures precision in a different way. It computes the mean of Average Precision values among a set of queries. The higher value, the better. The Average Precision (AP) value of a single query is:

$$AP = \sum_{k=1}^{M} \frac{P(k) \times pos(k)}{number\ of\ positive\ instances} \tag{5}$$

Here, k is the rank, M is the number of ranked files and $pos(k)$ is a binary indicator of relevance. $P(k)$ is the percentage of correctly reported buggy files among the top k results, and $pos(k)$ is a binary indicator for whether or not the k^{th} file is buggy. For example, if 5 documents are retrieved, and the 3^{rd} and 5^{th} are buggy, then AP is $(\frac{1}{3} + \frac{2}{5})/2 = 0.37$.

- **Normalized Discounted Cumulative Gain (NDCG)** is a widely used metric in recommendation systems [51, 64]. The basic concept of this metric is to calculate the relative difference between the recommended ranking and the ideal ranking. NDCG is defined as follows:

$$NDCG = \frac{DCG}{IDCG}, (DCG = \sum_{i=1}^{n} \frac{2^{relevance_i} - 1}{log_2(i+1)}) \tag{6}$$

where $relevance_i = 1$ if the i-th source code file is related to the fault location, and $relevance_i = 0$ otherwise. IDCG is the ideal order of DCG, which means all the faulty files are ranked higher than the unrelated files. For example, if an approach recommends three files in which the 3^{rd} file is error-related, the results are represented as $\{0, 0, 1\}$, whereas the ideal recommendation is represented as $\{1, 0, 0\}$. For the given example, DCG value is 0.5 and IDCG vlaue is 1.0, then NDCG value is $\frac{0.5}{1.0} = 0.5$. NDCG value ranges from 0.0 to 1.0 and is correlated to the MAP metric as it also evaluates the position of ranked items.

4.3 Research Questions

In our experiment, we investigated the following five research questions:

- **RQ1** How effective is UniLoc to locate bugs for CI failures? Although fault localization research is an active research area for over a decade, prior works [11, 16, 58, 79] mostly focus on source code fault localization. In our work, we considered both source code and build script for fault localization. Therefore, it is important to measure the effectiveness of UniLoc in comparison with existing approaches. To understand whether UniLoc works better than naive approach of file names mentioned in build error log file and existing tools, we will apply our proposed UniLoc, file name mentioned in error log file and baseline IR-based techniques: BLUiR and Locus to the same data set.
  Analysis Result: Our findings show that that UniLoc outperformed Baseline1 (file name mentioned in log file), Baseline2 (BLUiR) and Baseline3 (Locus) for all metric, specifically higher MAP and NDCG value indicate that UniLoc works better for different types of CI failure.
- **RQ2** What is the impact of recent change based file ranking for build fault localization? As build failures are usually caused by recent commits, it might be a natural intuition that faults are in the recently changed files. But our analysis finds that for many cases, fixes are in files other than recently changed files. To make a quantitative comparison, we compare UniLoc with an approach based on the changed files in the file-inducing commit.
  Analysis Result: Our findings show that change history or reverting-based approach has limited capability of FL considering wide-range of CI failure types. Overall, the performance of the approach is lower than UniLoc.
- **RQ3** How sensitive is UniLoc to different parameter settings and strategies applied? To improve the performance of UniLoc, we developed different techniques such as query optimization, search space optimization, etc., we need to have an evaluation of the usefulness of these techniques. To understand how UniLoc works with different configurations, we changed the parameter values and also created variant approaches by disabling one technique at a time.
  Analysis Result: Our findings show that UniLoc is sensitive to both parameter lt—the similarity threshold between two lines of build information and α—the exponential value used to improve file ranking. Our

approach worked best with $lt = 0.9$ and $\alpha = 0.1$. Apart from that, among the applied techniques: query optimization, search space optimization, and file ranking optimization, search space optimization contributed most to improve UniLoc performance.

- **RQ4** How effective is our approach for failures to be repaired in source code only, build script only, and both? As shown in Table 4.1, CI failure fixes can be in source code, build script, or both. Prior research works [58, 73] considers only source code. Since UniLoc targets both source code and build script, we further measure the effectiveness of UniLoc on failures fixed at different locations.
  Analysis Result: Baselines' performance varies a lot for different types of fixes, but UniLoc has a more robust and balanced performance among all three types of fixes and shows overall the best performance.
- **RQ5** How effective is our approach for different type of CI failures? A CI failure can be a test failure or a non-test failure. As characteristics of test failure and non-test failure might be different, measuring the performance of UniLoc for test failure and non-test failures can be useful to showcase the effectiveness of UniLoc for different types of failures.
  Analysis Result: For both test failure and non-test failure UniLoc performs better than all three baselines. However, UniLoc works more effectively on non-test failure.

4.4 Results

RQ1: Effectiveness of UniLoc. To understand the comparison between UniLoc and existing approaches, we applied UniLoc, file name mentioned in $PART'_f$(**Baseline1**) and state-of-the-art IR-based FL techniques BLUiR [58](**Baseline2**) and Locus [73](**Baseline3**) to our dataset. Since $PART'_f$ mentions error file names, so during fault localization file ranking for each file names mentioned in $PART'_f$ gets one point and other files get zero. This comparison will help us know whether our proposed approach works better than a keyword-based approach that locates failures based on file names mentioned in $PART'_f$ part. Apart from that, we compared UniLoc with BLUiR [58] and Locus [73]—state-of-the-art source-code-oriented IR-based FL techniques studied in recent studies [22, 39]. As BLUiR is not publicly available, we reimplemented BLUiR with default configuration parameters and structural code entities mentioned in the paper. BLUiR uses bug reports as queries and searches source code for bugs, to facilitate comparison, we extended the tool in two ways. First, instead of feeding in a bug report, we used the refined failure-relevant part $PART'_f$ as an input. We observed that CI logs are very large (typically more than thousands of lines) compared to bug reports and contain unrelated information such as downloading dependency, CI server-specific information, etc. As a result, applying the full log will affect baseline approaches' performance. In fact, we utilized full log and $PART'_f$ to BLUiR and observed that in terms of MMR and MAP metric performance degrades 17.13% and 22.79%, respectively, if we use full log. Moreover, a recent work [59] on CI configuration correctness also observed that the log error part contains important terms related to CI failure. So, rather than utilizing the full logs, we used $PART'_f$ for baseline FL techniques. Second, for source code, we provided AST entities to BLUiR and for build script we provided all the build script contents as text to BLUiR. For evaluation, we utilized publicly available existing Locus implementation [9]. However, we transformed our data to Locus compatible form with a data transfer process and modified the Locus data input process to allow analyzing multiple projects in batch mode. Like BLUiR, Locus also uses bug reports as queries and searches source code for bugs. Locus utilizes bug open date and fix date to extract change history based on the dates. For applying Locus to localize CI failures, we used the refined failure-relevant part $PART'_f$ as the bug description, the commit date that generated CI failure as bug open date and commit date that fixed CI failure as bug fix date. As CI failures do not have any failure summary, we kept Locus bug summary empty. Note that the identification of $PART'_f$ is based on our technique described in Section 3.1, so the baseline approaches already

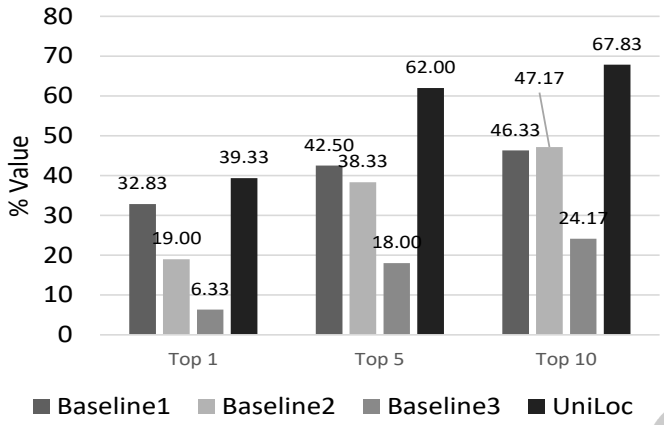

Fig. 4. Top-N Comparison between Baselines and UniLoc

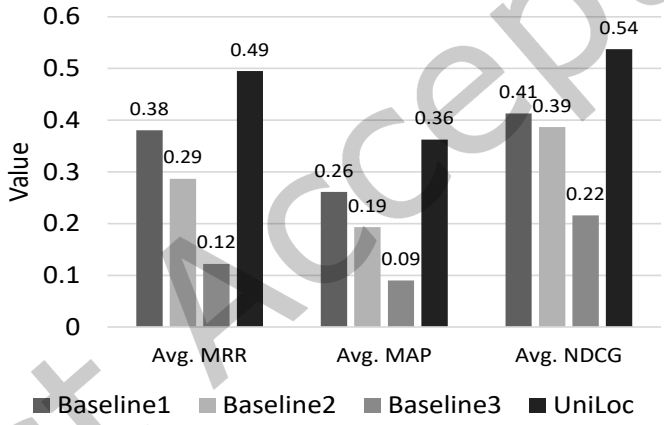

Fig. 5. MRR, MAP and NDCG Comparisons between Baselines and UniLoc

partly take advantage of our query optimization technique (directly using the whole build log as the query will result in much worse results similar to random file selection due to noises in the log).

Figure 4 and Figure 5 present the effectiveness comparison between baselines and UniLoc. According to Figure 4, Baseline1's Top-1, Top-5, and Top-10 values are 31.33%, 39% and 42.88%. While Baseline2's Top-1, Top-5, and Top-10 values are separately 19%, 38.33%, and 47.17%. At the same time, Baseline3's Top-1, Top-5 and Top-10 values are 6.33%, 18.00% and 24.17% respectively. In comparison, UniLoc's Top-1, Top-5, and Top-10 values are 39.33%, 62%, and 67.83%. UniLoc largely outperformed Baseline1, Baseline2 and Baseline3 by recommending more buggy files in the Top-N results. For Baseline1, even the result was good for Top-1 as for some build failures like compilation error or static analysis error it mentions file name in error log. But for complex cases where file names are not directly mentioned or fix is in different file location then Baseline1 does not work well. As a result for Top-10 it's performance is less effective than Baseline2, Baseline3 and UniLoc. Even though

Table 5. Effectiveness Comparison between Change History Based Approach and UniLoc

	Top 1	Top 5	Top 10	MRR	MAP	NDCG
Change History Based Approach	27.33%	51.50%	60.50%	0.38	0.28	0.46
UniLoc	39.33%	62.00%	67.83%	0.49	0.36	0.54

Baseline2 reviewand Baseline3 considers all the source files and build files for FL similarity analysis, Baseline2's and Baseline3's performances are lower than UniLoc. A possible reason for the lower performance of Baseline2 is that it is considering all files with the same weight. On the other hand, Baseline3 considers all prior all changes histories to localize faults rather than only recent changes that triggers the CI failure. But for CI failure, failures are generated by recent changes. So, recent change history does have a high impact on CI fault localization and prior FL techniques [35, 70] also utilize change history to improve the performance. In Figure 5, the Baseline1 technique achieved 0.38 MRR, 0.26 MAP and 0.41 NDCG, Baseline2 achieved 0.29 MRR, 0.19 MAP and 0.39 NDCG, while Baseline3 approach achieved 0.12 MRR, 0.12MAP and 0.22 NDCG. Our proposed approach UniLoc achieved 0.49,0.36 and 0.54 as MRR, MAP and NDCG, respectively, so UniLoc shows higher effectiveness than the baselines.

Specifically, in Figure 5, UniLoc has wider value ranges of both MAP and NDCG than baselines. Since MAP and NDCG metric considers all files ranking rather than one best file ranking, it means that UniLoc's effectiveness can vary on different CI failures.

Finding 1: *UniLoc outperformed Baseline1, Baseline2 and Baseline3 for all metric, specifically higher MAP and NDCG value indicate that UniLoc works better for different types of CI failure.*

RQ2: Recent Change History Based Fault Localization. We observed that 41.14% of CI failure fixes contain at least one line of change revert in source code or build script. So, we were curious about pure history-dependent fault localization. For Change History Based FL, we consider the changes in the failure-inducing commit. Instead of calculating similarity for these files, we gave the final score as 1.0 for the files changed in the failure-inducing code commit. For other files, the final score is assigned as 0.0. With this change heuristic driven approach, we calculated Top N, MRR, MAP and NDCG metrics. We also compared the Change History Based approach with UniLoc. Table 5 shows the effectiveness comparison in between the change based approach and our proposed approach. The Change History Based approach achieves 0.38 MRR, 0.28 MAP and 0.46 NDCG (compared to 0.49 MRR, 0.36 MAP and 0.54 NDCG of UniLoc). It also achieves 27.33%, 51.5%, and 60.5% for Top-1, 5, and 10 metrics (compared to 39.33%, 62.00%, and 67.83% of UniLoc).

Finding 2: *UniLoc provides better performance over Change History or Reverting Based approach for CI fault localization.*

RQ3: Sensitivity to Parameters and Strategies. There are two parameters used in UniLoc: lt—the similarity threshold between two lines of build information, and α—the exponential value used to improve file ranking. To explore UniLoc's sensitivity to these parameter settings, we tried lt={0, 0.5, 0.6, 0.7, 0.8, 0.9} and changed α between 0.0 and 0.9, with 0.1 increment. As shown in Figure 6, UniLoc obtained the highest MRR value when lt=0.9 and α=0.1.

Finding 3: *UniLoc is sensitive to both parameters: lt and α. It worked best when $lt = 0.9$, $\alpha = 0.1$.*

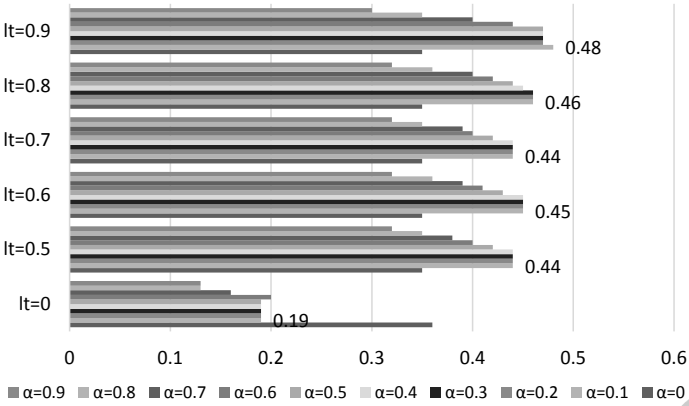

Fig. 6. MRR for Different Parameter Value on Tuning Dataset

Table 6. Effectiveness Comparison between variant approaches, Baselines, and UniLoc

Approach Name	Top 1	Top 5	Top 10	MRR	MAP	NDCG
Baseline1	32.83%	42.50%	46.33%	0.38	0.26	0.41
Baseline2	19.00%	38.33%	47.17%	0.29	0.19	0.39
Baseline3	6.33%	18.00%	24.17%	0.12	0.09	0.22
V1—Without Query Optimization	33.50%	59.50%	66.16%	0.44	0.33	0.51
V2—Without Search Space Optimization	13.16%	27.00%	35.00%	0.21	0.14	0.50
V3—Without File Ranking Optimization	16.67%	36.50%	42.83%	0.26	0.19	0.37
UniLoc	39.33%	62.00%	67.83%	0.49	0.36	0.54

There are three strategies applied in UniLoc: S1—query optimization (Section 3.1), S2—search space optimization (Section 3.2), and S3—file ranking optimization (Section 3.4). To understand how different strategies influence UniLoc's effectiveness, we also created three variants of our tool: V1—a variant without applying S1, V2—a variant without S2, and V3—a variant without S3.

Table 6 shows the effectiveness comparison between variants, Baselines, and UniLoc. According to the table, without search space optimization(S2) and file ranking optimization(S3) UniLoc worked worse than baselines. Among S2 and S3 approaches, S2 that is build dependency analysis and AST entity use plays the most important role for performance improvement. Among the variants, V1 worked much better than V2, which implies that S2 and S3 are more effective than S1. V2 has less metric values than V1 and V2, meaning that S2 is much more important than the other two strategies. Uses of changed files in failure-inducing commits are also important for UniLoc's performance improvement.

Finding 4: *Query optimization, search space optimization, and file ranking optimization all help with fault localization, while search space optimization boosts effectiveness the most.*

Table 7. Effectiveness Evaluation for Different Fix Types

Fix Type	Approach Name	Fail. Cnt.	Top 1	Top 5	Top 10	MRR	MAP	NDCG
Source Only	**Baseline1**	430	**30.47**	38.84	42.09	0.35	0.25	0.39
	Baseline2		2.56	19.53	29.07	0.11	0.10	0.28
	Baseline3		3.49	13.26	18.14	0.09	0.06	0.14
	UniLoc		30.00	**53.26**	**59.30**	**0.41**	**0.31**	**0.48**
Build Only	**Baseline1**	78	26.92	34.62	35.90	0.31	0.29	0.39
	Baseline2		50.00	**82.05**	**89.74**	0.64	0.60	0.70
	Baseline3		12.82	33.33	47.44	0.22	0.19	0.50
	UniLoc		**65.38**	**82.05**	87.16	**0.74**	**0.67**	**0.75**
Both	**Baseline1**	92	48.91	66.30	75.00	0.57	0.28	0.52
	Baseline2		**69.57**	**89.13**	**95.65**	**0.80**	0.30	0.58
	Baseline3		14.13	27.17	32.60	0.21	0.14	0.31
	UniLoc		60.87	85.87	91.30	0.71	**0.34**	**0.60**

RQ4: Effectiveness for Different Types of Bug Locations The bug locations of CI failures may be in source files, build scripts, or both types of files. We further clustered UniLoc's evaluation results among these three kinds of scenarios. As shown in Table 7, among the 600 evaluated CI failures, 430 failures were fixed by only source code changes, 78 failures were fixed by only modifying build scripts, and 92 failures were fixed by changes in both types of files. These number already shows the complexity of CI failure fixes. In Table 7, we can see that all approaches perform better when the fixes are in build scripts or in both types of files, maybe because there are fewer build script files than source files. Furthermore, three baselines perform very differently in different types of fixes, but UniLoc is more balanced and always has the best performance score in terms of MAP and NDCG. As MAP and NDCG metric considers all faulty file's ranking for performance calculation, UniLoc shows a more robust FL ranking considering all faulty files. In fact, in the evaluation dataset, the average number of files modified to fix CI failures is 6.8 files per failure, with an average of 7.32 source file modification if it requires source-related modification and an average of 1.48 build script modification if it involves build-related failure. Although, in most cases build related CI failure requires one or two files modification from the average of 9 build script files, it is always difficult to determine whether any build-script change is required. Furthermore, identifying faulty build scripts is more challenging than identifying faulty source files due to the high-level abstraction of statements in build scripts, latent dependencies between build scripts and source code and among build scripts themselves, and lack of tool support for build-script debugging.

Moreover, among 600 CI failure, 347 failures required more than one file modification(s) to fix the CI failure. This brings in the necessity of more robust FL tools like UniLoc to localize CI failure root causes. Apart from that, since it is not possible to know the type of fixes in advance, UniLoc's balance and robust performance will help it achieve the expected performance in most cases.

More surprisingly, **Baseline2** (BLUiR) performed better on build script fixes, but **Baseline1** (querying file names in $PART'_f$) performed better on source code fixes, which is different from our expectations. **Baseline3** (Locus) preformed worst for source related CI failures. After a detailed investigation, we found that, even after our query optimization, the build logs still contain some noises which look similar to AST elements of the build script, so they misled BLUiR even when the actual fixes required are in the source file(s). This result is consistent with UniLoc without optimizations $V2$ or $V3$, as shown in Table 6. This shows that even better query-optimization techniques are still required for applying IR-based FL approaches to CI scenarios. For build scripts, BLUiR performs better because build-related terms from specific build script(s) are dominating the build logs, so it simply ranked all of them higher, and thus had higher Top-10 coverage (note that their Top-1 coverage

is much lower than UniLoc). But if the fix requires to change in multiple build script with module dependency, then in those cases Baseline 2 cannot perform well. As a result, even for many cases, Top-N (considers one file only) metric result is better, but in terms of MAP and NDCG metric, Baseline 2 performance is lower than UniLoc. In contrast, **Baseline1** performs better on source-only fixes. Our further investigation (presented in Table 8) shows that its high performance mainly comes from source-only fixes of non-test failures, which are mainly compilation errors and code-style errors. Since file names are often provided in such non-test failures, it is no wonder that **Baseline1** performed very well on them. But in many cases fixing compilation errors and code-style errors might require changes in other files (due to dependency) that are not mentioned in failure log. For those cases, **Baseline1** might show promising results in terms of Top-N and MRR metric, but for MAP and NDCG metric **Baseline1**'s performance is lower than others. The analysis also shows that looking at the error log for the faulty file might not be sufficient to solve CI failures, and the approach can suffer from missing fault-related files in fault localization. For fixes with both file change types, Baseline2 showed promising results in terms of Top1, Top5, Top10 and MRR metrics. These four matrices consider first file matching rather than all faulty file matching. Among both file type-related fixes, many of the fixes are related to plug-in, compilation and dependency-related failure where the failed log clarifies which file with configuration or code entity(s) generated errors. However, fixing those failures, in most cases, requires changes in files with matching entities, and requires changes in some other related files whose names or entities are absent in the failure log. Baseline2 can rank the first matching file in the higher position but cannot identify other related files due to lack of dependency information and recent change information. Example 5 shows such a build failure, where the build log explicitly mentioned entities related to */sandbox/build.gradle* file; however, while fixing developer changed this file, as well as four other .java source files. In this case, Baseline2 can rank */sandbox/build.gradle* in a higher position, but the approach cannot prioritize other related files. As a result, in terms of MAP and NDCG metric, UniLoc outperforms Baseline2 and other approaches. In the case of **Baseline3**, the tool performed poorly as the approach considers all priors commit history and change hunks for FL. However, prior commits might be related to bug fixes, feature addition, and CI fixes and it cannot differentiate CI fixes from other kind of code modifications. Since the number of other kinds of modifications(e.g., bug fix, feature addition) are much more frequent than CI fix modifications, in most cases, FL rankings generated by this approach are less relevant to CI failures.

Example 5 Failure Segment that Requires Change in Both File Type *(jphp-compiler/jphp)*

```
CI Log Part for Commit ID: cb98af7
_____________________________________________
...
FAILURE: Build failed with an exception.
Build file '/home/~/sandbox/build.gradle' line: 13
* What went wrong:
A problem occurred evaluating project ':sandbox'.
&gt; Failed to apply plugin [id 'php']
   &gt; Plugin with id 'php' not found.
Run with --stacktrace option to get the stack trace. Run with --info or --debug option to get more log output.
BUILD FAILED
...
```

Finding 5: *Baselines' performance varies a lot for different types of fixes, but UniLoc has more robust and balanced performance among all three types of fixes.*

Table 8. Effectiveness for Different Failure Types

Approach Name	Only Source Change			Only Build Script Change			Both Change			Overall		
	Test Failure											
	Cnt. 222			Cnt. 21			Cnt. 31			Cnt. 274		
	MRR	MAP	NDCG	MRR	MAP	NDCG	MRR	MAP	NDCG	MRR	MAP	NDCG
Baseline1	0.19	0.14	0.30	0.06	0.05	0.19	0.23	0.13	0.40	0.18	0.13	0.30
Baseline2	0.09	0.08	0.27	0.58	0.51	0.63	**0.69**	0.34	0.60	0.19	0.14	0.34
Baseline3	0.11	0.08	0.15	0.25	0.22	0.57	0.33	0.27	0.40	0.14	0.11	0.21
UniLoc	**0.35**	**0.27**	**0.45**	**0.66**	**0.58**	**0.66**	0.67	**0.40**	**0.64**	**0.41**	**0.31**	**0.49**
	Non-Test Failure											
	Cnt. 208			Cnt. 57			Cnt. 61			Cnt. 326		
	MRR	MAP	NDCG	MRR	MAP	NDCG	MRR	MAP	NDCG	MRR	MAP	NDCG
Baseline1	**0.53**	**0.38**	0.49	0.40	0.37	0.46	0.75	**0.35**	**0.60**	0.55	0.37	0.50
Baseline2	0.14	0.12	0.31	0.65	0.64	0.72	**0.85**	0.27	0.57	0.36	0.23	0.43
Baseline3	0.06	0.04	0.14	0.21	0.18	0.48	0.15	0.08	0.26	0.10	0.07	0.22
UniLoc	0.47	0.35	**0.52**	**0.76**	**0.71**	**0.78**	0.72	0.31	0.58	**0.57**	**0.40**	**0.58**

RQ5: Effectiveness of Different Types of CI Failures. CI failures can be triggered by test failures or non-test failures. We also clustered UniLoc's evaluation results among these two kinds of failures and presented the results in Table 8. Table 8 presents the performance of UniLoc and baseline approaches using MRR, MAP and NDCG metrics. Since both Top-N and MRR metric utilizes only the first matched buggy file's ranks among all buggy files to calculate the performance, we believe that MRR reflects the performance considering the first buggy file match. As a result, we did not present the performance evaluation with Top-1, Top-5 and Top-10 metrics for this analysis. Table 8 shows one important insight that 18.97% of test failure fixes require accompanying changes in the build script. Although prior research works [69, 85] on fault localization identifies that spectrum-based fault localization (SBFL) works better for Test Failures, based on test failure fix statistics (see Table 8), existing SBFL techniques might not work for test failures that require build script change. Moreover, for SBFL, running these test cases on an instrumented version of the faulty program may not track build script execution traces. Since it is not possible to know the type of fixes in advance, UniLoc does have an advantage over SBFL techniques to localize faults in a balanced way.

In Table 8, we can observe that all approaches except **Baseline3** performed better on non-test failures than test failures, which is reasonable as the latter can be more complicated and involve more files. Baseline3 Locus was mainly optimized to detect test failures. However, in many cases in CI environment, test failures happen due to missing dependency or run-time class binding that generates exceptions and can fail a test. As a result, even though Baseline3 performed better for test failures rather than non-test failures, the overall performance of Baseline3 is low. At the same time, for test failures such as Example 6 where the file name is mentioned in the build log, Baseline1 can find the first faulty file based on the log but cannot identify dependent files that are also required to fix. To fix Example 6 failure, the developer needs to make changes in six files, and only one of them is mentioned in the build log. Baseline2 can find those faulty files, but its ranking is not optimized due to the lack of build dependency information, as well as change history information. Overall for test failure, UniLoc outperformed all baseline approaches for all the metrics. In terms of NDCG, the improvement over Baseline1 and Baseline2 are 38.77% and 30.61%, respectively.

Apart from test failures, there can be non-test failures due to configuration errors, compilation issues, static analyzers (e.g., CheckStyle, Lint), etc. For non-test failures involving only source fix, Baseline1 shows better performance. So we did an analysis for performance improvement and observed that compilation errors and

Example 6 Test Failure Segment *(thatJavaNerd/JRAW)*

```
CI Log Part for Commit ID: 5e45bab
_____________________________________________
...
/home/~/auth/OAuth2Test.java:111: error: exception ApiException is never thrown in body of corresponding try
     statement
       } catch (NetworkException | ApiException e) {
:compileTestJava FAILED
FAILURE: Build failed with an exception.
...
```

CheckStyle errors are common in this category. In such failures, all or many error files are mentioned in the build log , so it works better. Example 7 shows such a non-test build failure where the faulty file is directly mentioned in the build log and the developer made modifications in the mentioned file. But for the cases where changing a class file required further changes in its parent class, BaseLine1 cannot localize all the faulty files. For non-test failures involving build script only, UniLoc outperforms Baseline1 and Baseline2 due to optimized use of build script ASTs. Baseline1 shows a surprisingly good result for non-test failures with both source and build source file change. So we analyzed the cases where Baseline1 outperforms UniLoc. From our analysis, we observed that among these 61 failures, 27 failures were from the same project (BuildCraft/BuildCraft). Then we analyzed the commits (Example Commit ID:d236d08) of these 27 failures and observed that after each build error fix, the developer updated the version number in root `build.gradle` file, which is not related to the build error but related to Checkstyle convention. Since UniLoc optimizes ranking with precise build dependency information and AST optimization, such unrelated `build.gradle` is ranked lower. In contrast, `build.gradle` almost always shows up in the build log and it is ranked highest among all files by default because it is in the root folder and starts with 'b'. Besides, CheckStyle usually reports all the file names in the build log that violates stylistic rules. So Baseline1 performs very well on these failures. But this type of CI failure is uncommon and project-specific. Even involving these 27 failures, UniLoc's performance for non-test failures involving both file type changes is only 3.33% lower than BaseLine1's in terms of NDCG metric. Overall for non-test failure, UniLoc outperformed both BaseLine1 and BaseLine2 in terms of three metrics MRR, MAP and NDCG. The results show that UniLoc performs better than baselines for both types of failures.

Example 7 Non-Test Failure *(BuildCraft/BuildCraft)*

```
CI Log Part for Commit ID: 20f6900
_____________________________________________
...
:checkstyleMain[ant:checkstyle] /home/~/ItemLaserTable.java:11:8: Unused import - java.util.List.
[ant:checkstyle] /home/~/ItemLaserTable.java:14:8: Unused import - net.minecraft.entity.player.EntityPlayer.
..
FAILURE: Build failed with an exception.
Execution failed for task ':checkstyleMain'.
* Try:
Run with --stacktrace option to get the stack trace. Run with --info or --debug option to get more log output.
BUILD FAILED
...
}
```

Finding 6: *UniLoc works more effectively on non-test failures and performs better than all three baselines on both test failures and non-test failures.*

5 THREATS TO VALIDITY

One potential internal validity of our evaluation is the ground truth we considered to resolve the CI failures may contain code changes other than build fix like code enhancement, refactoring etc. Actually since changes in one code commit may be dependent on each other (i.e., a partial commit may not compile or passes tests), there does not exist a clear definition for the relevant part in the repairing commit. We assumed that the one we extracted from the project repository was the best one, as it was the actual developer fix. To reduce the threat, we only considered CI build instances with failed status to passed status with modification of build script or source code in one single commit. Prior research efforts [37, 55, 69] on FL also used the difference of pre-fix and post-fix code commits as the ground truth for evaluation. Actually, among the 600 evaluated fixes, 253 (42.16%) fixes touched only one file (so their ground truth is fully precise for our evaluation) and the rest of the 347 (57.83%) fixes touched multiple file change, so only them might be affected by the threat. The major external threat to our evaluation is that we evaluated UniLoc for only Gradle based project with Java as a programming language. We tried to make our approach generalized so that it can be applied to other build management tools and programming languages. To reduce this threat, we plan to apply our approach to other popular build systems and programming languages. Last of all, our FL technique can identify faults in file level, which could be coarse-grained for the practical usability of the tool. However, considering the heterogeneity of CI failures and the different nature of CI failures, the file-level FL technique can be helpful for developers. To mitigate such threat, we plan to extend the FL technique to have more fine-grained such as block-level or statement-level localization capability.

6 DISCUSSIONS

This section discusses the implications of our results and observations from our study on FL. It further outlines some future research directions.

6.1 Implications

Implications of the Dataset Observations. A significant amount of research effort has been devoted to CI in identifying barriers to adopt CI and optimize the workflow of CI. Prior study on CI workflow [75] identified that increased build complexity is the main reason to abandon CI work process. Hilton et al. [31] identified that troubleshooting CI build failures as the top barrier when developers using CI. To overcome such barrier, several research works on CI build failure have been conducted. Vassallo et al. [68] did an analysis of CI build failure on 349 Java OSS projects and 418 projects from a financial organization. They categorized CI build failure into 20 categories and identified that testing failures and release preparation failures are the topmost reasons for CI failure. A recent study [26] on build breakage data identified that 33% of the build breakages are due to environmental factors, 29% are due to errors in previous builds, and 9% are due to build jobs. In our study, we also categorized 700 build failures into two broad categories: i) Test Failure and ii) Non-Test Failure. Among these failures, 45% CI failures are test failures and 55% failures are non-test failures. Apart from that 17% of test failure require to have fixed in build script and 24% overall failures need to have fixed in build script. These are important empirical evidence for tool builders to work on the heterogeneity nature of CI failures and the requirement of tool support for source code and build script.

Implications of the CI Failure Debugging. Fault Localization has been a widely explored research area over the decades. IRFL-based fault localization techniques [58, 71, 73] are mostly using bug reports to find faulty code locations. While SBFL based techniques [11, 79, 80] mostly rely on test case execution results to find source

code fault locations and require instrumentation support. Even after the wide advancement of FL techniques, a prior empirical study on CI [31] identified the need for debugging assistance for CI workflow. Moreover, prior studies [20, 69] identified that SBFL and IR-based fault localization has limited usability in the real-world development workflow due to execution overhead, limited bug context information, inaccurate ranking, etc. Considering the limitations, our proposed approach aligns with the CI workflow and can localize CI faults without instrumentation overhead. Also, UniLoc can localize faults in both source code and build script, which is not supported by prior FL techniques. Our empirical analysis also suggests that UniLoc outperforms existing IR-based FL techniques for test failure and non-test failures. UniLoc can be a more effective tool support to meet the needs of debugging assistance in CI. Our approach was evaluated on real-world CI failures from large-scale open-source projects, suggesting that the UniLoc can be useful in real-world development scenarios. Apart from that, UniLoc is one of the first in its kind for fault localizing build script, which is limited in the count but different than source code in terms of abstraction, domain knowledge and limited tool support for debugging. Considering the heterogeneity of CI failure involving source code and build script and limited support of debugging build script, our proposed approach with IR-based FL in file-level granularity can be useful for developers to debug CI failure. The work can be the basis for further research on the usability of such tool support and more fine-grained unified fault localization tools, such as at block level, given that a widely accepted definition of blocks in build scripts can be developed.

6.2 Research Directions

Build Tools Other than Gradle. In this research work we only considered CI failures of projects using Gradle as their build management tool. The major Gradle-specific part of UniLoc is our build dependency module. Like Gradle, other popular build systems also provide support for multi-module build. Ant [1] provides multi-module build with dependency information with the help of Ivy [2]. In Maven [4], mechanism to handle multi-module build is called Reactor. With the help of Reactor, Maven can also define project dependency. Apart from that, Ant and Maven also provide build log with rich source of information like build status, fail information, compilation issue etc. Moreover, Ant and Maven build failures are also available in TravisTorrent dataset. So, our approach can be applied to other build management tools by extending our build configuration and build script analyzer, as well as be evaluated on the TravisTorrent dataset.

IR-based vs. Spectrum-based Fault Localization. Compared with spectrum-based fault localization, IR-based fault localization is often less precise due to the lack of runtime information. In contrast, IR-based approach can be applied without code instrumentation. In the scenario of continuous integration, even if code instrumentation support does exist, it cannot be always turned on due to the high overhead. So due to the urgency of resolving CI stalls, an IR-based approach can be very helpful with the initial assignment of bugs to a proper developer, and the developer's initial investigation. Furthermore, as illustrated in multiple examples in this paper, CI failures often involve multiple types of files (e.g., source files, build scripts) and their dependencies. In such cases, code instrumentation on one file type may miss root causes of failures, while a comprehensive code instrumentation support can be difficult to implement. Apart from that, in some CI practices, CI servers queue multiple commits into a single commit to optimize integration time and testing time [8]. In those case applying, applying SBFL on smaller sub-commits can be impractical due to resource and time limitation and IR-based approach can be more efficient on smaller sub-commits to identify faulty files.

7 RELATED WORKS

7.1 Automatic Bug Localization

Automatic bug localization has been an active research area over the decades [41] [84]. Automatic bug localization techniques can be generally divided into two categories: i) dynamic approaches and ii) static approaches. Dynamic

fault localization [53] requires execution of programs and test cases to identify precise fault location. Dynamic fault localization techniques need pre-processing of the code or underlying platform, as well as precise reproduction of the failure. Among the dynamic fault localization techniques, spectrum based fault localization (SBFL) [11] is the most prominent technique. SBFL usually depends on suspicion score based on program elements executed by the test cases. Tarantula [33] is the early research work on SBFL and subsequent researchers are working to improve the accuracy of the localization technique. Ochiai [10] uses different similarity co-efficient to find more accurate fault localization. Xuan and Monperrius proposed Multric [79], which combines learning-to-rank and fault localization techniques for more accurate localization. Savant [16] uses likely invariant with learning-to-rank algorithm for fault localization. Küçük et al. [36] proposed a novel approach that combines the causal inference from code and coverage information. Sarhan et al. [60] developed a fault localization tool for python based on exisitng spectrum-based approaches. Most recently, Lou et al. [43] and Li et al. [40] use representation learning on code dependencies and run-time code coverage to predict the failure-causing statements. Meng et al. [50] further enhanced these techniques by incorporating knowledge from historical bugs and code from other software projects. Since the software building process lacks test cases and instrumentation of all building scripts in various forms can be challenging, the dynamic localization techniques mentioned above cannot be easily applied to faults in build scripts and configuration files.

Static fault localization techniques do not require test cases and execution information. Static fault localization depends on static source code analysis [83] [23] or information retrieval based approaches [69] [84]. Lint[32] is one of the first tool to find fault in C programs. FindBug [15] and PMD [5] are prominent static code analyzer for Java source code. Lots of IR-based approaches [84] [58] have been proposed by the researchers for fault localization. BugLocator [84] performs bug localization based on revised VSM model. Saha et al. [58] proposed BLUiR considering source code structure for IR-based fault localization. Recent work on fault localization Locus [73] utilizes change history for fault localization. Since static fault localization does not require execution environment and test cases, we applied IR-based fault localization technique for build fault localization. In our approach, we adopted build script analysis, source code AST and also recent change history for locating build fault from build log information.

7.2 Fault Localization Supporting Automatic Program Repair

Over the last few years Automatic Program Repair [34] [27] is gaining popularity. GenProg [27] uses Genetic Programming(GP) for automatic patch generation. RSRepair [54] performs random searching for generating a path. To reduce searching from existing code, PAR [34] uses predefined fix patterns to generate a patch for a new bug. Apart from search-based or template-based patch generation, machine learning and probabilistic models are also getting popularity for automatic program repair. Prophet [42] uses a probabilistic model to generate a new patch. Van Tonder and Le Goues [66] applied separation logic for automatic program generation. While Wen et al. [72] used AST context information for better program repair. Automatic program repair techniques are also getting popular for automatic build repair. Recently Macho et al. [46] proposed BUILDMEDIC to repair Maven dependency failure. HireBuild [30] uses a history driven approach for Gradle build script repair. For all these automatic repair works, one of an integral part of the repair is fault localization. As discussed in 7.1, there are different approaches for bug localization. For automatic build repair, previous works consider only build script for their repair target. But build failure can be generated for source code, build script or both. So, apart from assisting developers for fixing build failure, build fault localization can be useful for automatic build repair research work.

7.3 Build Script Analysis

With the growing popularity of build management tools and automatic build scripts, analysis of build script is also getting importance for software engineering research areas such as build repair, fault localization, build

target decomposition, migration of build configuration, etc. For build dependency analysis, Gunter [14] proposed a Petri-net based model. Adams et al. [12] proposed re(verse)-engineering framework MAKAO to keep build consistency in change revisions. MAKAO extracts dependency from build traces to generate build consistency. Recently Wen et al. [74] proposed BLIMP for build change impact analysis generated from the build dependency graph. Xia et al. [78] proposed a machine learning based model to predict build co-changes. While from source code change history, Macho et al. [45] proposed model to predict build configuration changes. McIntosh et al. [49] performed a large study to find relation in between build maintenance and build technology. SYMake [63] uses a symbolic-evaluation based technique to detect common errors in Make files. To improve software build process, McIntosh et al. [48] did a study on header file hotspots. On the study of building errors, Hassan et al. [28] performed empirical analysis on build failure hierarchy.

The most closely related work is fault localization of Make build script proposed by Al-Kofahi et al. [13]. They proposed MkFault to generate suspiciousness scores of Make statement for a build error. MkFault instrumented code to generate build traces. But in CI environment, instrumenting large code base might be costly in terms of time and resource. Apart from that MkFault only considers Make build script as source of build failure. But our analysis on real build error fix finds that build error can happen due to source code, build script or both. We also performed evaluation of our approach on a large dataset with different project configuration. Recently Sharma et al. [61] proposed an approach to identify bad smells in configuration files.

8 CONCLUSION AND FUTURE WORK

Most existing approaches(e.g., Locus [73], BRTracer [76], BLUiR [58]) in fault localization focus on test failures or bug reports and source code or other single type of files for fault localization. By contrast, there are much less research on the fault localization of build scripts and repair. A more realistic scenario in practice is that multiple types of failures happen simultaneously and can be ascribed to multiple types of files. Our analysis of localization CI failure with BLUiR and Locus suggests that the approaches can localize CI failures to a certain extent but are not fully optimized to localize CI failures. In this research work, we proposed the first unified fault localization approach that considers both source code and build script to localize the repair for continuous integration. Our approach works on top of classical IR-based approach with query and search space optimization based on build configuration and CI log analysis, and generates suspicion ranking of faulty files including both source code and build script. Our evaluation on 600 real CI failure shows that UniLoc can localize faulty files with MRR as 0.49, MAP as 0.36 and NDCG as 0.54, which outperforms baseline approaches for all types of failures.

In the future, we plan to implement file level build dependency graphs to filter out irrelevant files in search space. File-based build dependency graph with change history might help us reduce search space dramatically. Apart from that, we plan to apply more advanced IR-based searching approaches to find better ranking. Our experiment results show that query optimization is still a key challenge of applying IR-based FL approaches to CI scenarios, so we plan to develop more advanced techniques on query optimization. Moreover, we are planning to expand our fault localization approach to the source code and build script block level to assist developers and automatic repair approaches better. Finally, beyond source code and build scripts there are also other types of files to be involved during software repair, especially in other scenarios. For example, in the fault localization of web applications, we need to consider html files, css files, client-side JavaScript files and server side source code. We plan to adapt our technique to more scenarios with heterogeneous bug locations.

ACKNOWLEDGMENTS

This material is based in part upon work supported by National Science Foundation awards CSPECC-1736209, CCF-1846467, CCF-2007718, CNS-2221843, CCF-1845446, CCF-2006278 and CCF-2152819.

REFERENCES

[1] 2018. Ant. https://ant.apache.org/. Accessed: 2018-08-18.
[2] 2018. Ivy. http://ant.apache.org/ivy/. Accessed: 2018-08-18.
[3] 2018. Lucene. http://lucene.apache.org/. Accessed: 2018-08-18.
[4] 2018. Maven. https://maven.apache.org/. Accessed: 2018-08-18.
[5] 2018. PMD. https://pmd.github.io/. Accessed: 2018-08-18.
[6] 2019. An Introduction to CI/CD Best Practices. https://www.digitalocean.com/community/tutorials/an-introduction-to-ci-cd-best-practices.
[7] 2019. Why Continuous Integration Doesn't Work. https://devops.com/continuous-integration-doesnt-work/.
[8] 2022. Improving the Efficiency of CI with Uber-commits. https://digitalcommons.unl.edu/cgi/viewcontent.cgi?article=1126&context=computerscidiss. Accessed: 2022-10-27.
[9] 2022. Locus Implementation. https://github.com/justinwm/Locus/. Accessed: 2022-10-27.
[10] R. Abreu, P. Zoeteweij, and A. J. c. Van Gemund. 2006. An Evaluation of Similarity Coefficients for Software Fault Localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*. 39–46. https://doi.org/10.1109/PRDC.2006.18
[11] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. 2009. Spectrum-Based Multiple Fault Localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering (ASE '09)*. IEEE Computer Society, Washington, DC, USA, 88–99. https://doi.org/10.1109/ASE.2009.25
[12] B. Adams, H. Tromp, K. de Schutter, and W. de Meuter. 2007. Design recovery and maintenance of build systems. In *2007 IEEE International Conference on Software Maintenance*. 114–123. https://doi.org/10.1109/ICSM.2007.4362624
[13] Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N. Nguyen. 2014. Fault Localization for Build Code Errors in Makefiles. In *Companion Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE Companion 2014)*. ACM, New York, NY, USA, 600–601. https://doi.org/10.1145/2591062.2591135
[14] Nasreddine Aoumeur and Gunter Saake. 2004. Dynamically Evolving Concurrent Information Systems Specification and Validation: A Component-based Petri Nets Proposal. *Data Knowl. Eng.* 50, 2 (Aug. 2004), 117–173. https://doi.org/10.1016./j.datak.2003.10.005
[15] Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. 2007. Using FindBugs on Production Software. In *Companion to the 22Nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications Companion* (Montreal, Quebec, Canada) *(OOPSLA '07)*. ACM, New York, NY, USA, 805–806. https://doi.org/10.1145/1297846.1297897
[16] Tien-Duy B. Le, David Lo, Claire Le Goues, and Lars Grunske. 2016. A Learning-to-rank Based Fault Localization Approach Using Likely Invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis* (Saarbrücken, Germany) *(ISSTA 2016)*. ACM, New York, NY, USA, 177–188. https://doi.org/10.1145/2931037.2931049
[17] Brenda S Baker. 1999. Parameterized diff. In *Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms*. Society for Industrial and Applied Mathematics, 854–855.
[18] Len Bass, Ingo Weber, and Liming Zhu. 2015. *DevOps: A software architect's perspective*. Addison-Wesley Professional.
[19] M. Beller, G. Gousios, and A. Zaidman. 2017. TravisTorrent: Synthesizing Travis CI and GitHub for Full-Stack Research on Continuous Integration. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories*. 447–450.
[20] Tung Dao, Max Wang, and Na Meng. 2021. Exploring the Triggering Modes of Spectrum-Based Fault Localization: An Industrial Case. In *2021 14th IEEE Conference on Software Testing, Verification and Validation (ICST)*. 406–416. https://doi.org/10.1109/ICST49551.2021.00052
[21] Tung Dao, Lingming Zhang, and Na Meng. 2017. How Does Execution Information Help with Information-retrieval Based Bug Localization?. In *Proceedings of the 25th International Conference on Program Comprehension* (Buenos Aires, Argentina) *(ICPC '17)*. IEEE Press, Piscataway, NJ, USA, 241–250. https://doi.org/10.1109/ICPC.2017.29
[22] T. Dao, L. Zhang, and N. Meng. 2017. How Does Execution Information Help with Information-Retrieval Based Bug Localization?. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. 241–250. https://doi.org/10.1109/ICPC.2017.29
[23] Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson Murphy-Hill. 2017. Just-in-time Static Analysis. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. ACM, New York, NY, USA, 307–317. https://doi.org/10.1145/3092703.3092705
[24] Paul M Duvall, Steve Matyas, and Andrew Glover. 2007. *Continuous integration: improving software quality and reducing risk*. Pearson Education.
[25] Keheliya Gallaba, Christian Macho, Martin Pinzger, and Shane McIntosh. 2018. Noise and Heterogeneity in Historical Build Data: An Empirical Study of Travis CI. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. Association for Computing Machinery, New York, NY, USA, 87–97. https://doi.org/10.1145/3238147.3238171
[26] Taher Ghaleb, Daniel Costa, Ying Zou, and Ahmed E. Hassan. 2019. Studying the Impact of Noises in Build Breakage Data. *IEEE Transactions on Software Engineering* (08 2019), 1–14. https://doi.org/10.1109/TSE.2019.2941880
[27] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *IEEE Transactions on Software Engineering* 38, 1 (Jan 2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[28] F. Hassan, S. Mostafa, E. S. L. Lam, and X. Wang. 2017. Automatic Building of Java Projects in Software Repositories: A Study on Feasibility and Challenges. In *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. 38–47. https://doi.org/10.1109/ESEM.2017.11

[29] Foyzul Hassan and Xiaoyin Wang. 2017. Change-Aware Build Prediction Model for Stall Avoidance in Continuous Integration *(ESEM '17)*. IEEE Press, 157–162. https://doi.org/10.1109/ESEM.2017.23

[30] Foyzul Hassan and Xiaoyin Wang. 2018. HireBuild: An Automatic Approach to History-driven Repair of Build Scripts. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 1078–1089. https://doi.org/10.1145/3180155.3180181

[31] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. 2017. Trade-Offs in Continuous Integration: Assurance, Security, and Flexibility. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 197–207. https://doi.org/10.1145/3106237.3106270

[32] Stephen C Johnson. 1977. *Lint, a C program checker*. Citeseer.

[33] James A. Jones and Mary Jean Harrold. 2005. Empirical Evaluation of the Tarantula Automatic Fault-localization Technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering* (Long Beach, CA, USA) *(ASE '05)*. ACM, New York, NY, USA, 273–282. https://doi.org/10.1145/1101908.1101949

[34] D. Kim, J. Nam, J. Song, and S. Kim. 2013. Automatic patch generation learned from human-written patches. In *2013 35th International Conference on Software Engineering (ICSE)*. 802–811. https://doi.org/10.1109/ICSE.2013.6606626

[35] Sunghun Kim, Thomas Zimmermann, E. James Whitehead Jr., and Andreas Zeller. 2007. Predicting Faults from Cached History. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07)*. IEEE Computer Society, USA, 489–498. https://doi.org/10.1109/ICSE.2007.66

[36] Yiğit Küçük, Tim AD Henderson, and Andy Podgurski. 2021. Improving fault localization by integrating value and predicate based causal inference techniques. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 649–660.

[37] Tien-Duy B. Le, Richard J. Oentaryo, and David Lo. 2015. Information Retrieval and Spectrum Based Bug Localization: Better Together. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. ACM, New York, NY, USA, 579–590. https://doi.org/10.1145/2786805.2786880

[38] Jaekwon Lee, Dongsun Kim, Tegawendé F. Bissyandé, Woosung Jung, and Yves Le Traon. 2018. Bench4BL: Reproducibility Study on the Performance of IR-based Bug Localization. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. ACM, New York, NY, USA, 61–72. https://doi.org/10.1145/3213846.3213856

[39] Jaekwon Lee, Dongsun Kim, Tegawendé F. Bissyandé, Woosung Jung, and Yves Le Traon. 2018. Bench4BL: Reproducibility Study on the Performance of IR-Based Bug Localization. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Amsterdam, Netherlands) *(ISSTA 2018)*. Association for Computing Machinery, New York, NY, USA, 61–72. https://doi.org/10.1145/3213846.3213856

[40] Yi Li, Shaohua Wang, and Tien Nguyen. 2021. Fault localization with code coverage representation learning. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 661–673.

[41] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P. Midkiff. 2005. SOBER: Statistical Model-based Bug Localization. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Lisbon, Portugal) *(ESEC/FSE-13)*. ACM, New York, NY, USA, 286–295. https://doi.org/10.1145/1081706.1081753

[42] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. ACM, New York, NY, USA, 298–312. https://doi.org/10.1145/2837614.2837617

[43] Yiling Lou, Qihao Zhu, Jinhao Dong, Xia Li, Zeyu Sun, Dan Hao, Lu Zhang, and Lingming Zhang. 2021. Boosting coverage-based fault localization via graph-based representation learning. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 664–676.

[44] Lucia LUCIA, Ferdian Thung, David Lo, and Lingxiao Jiang. 2012. Are faults localizable? (2012).

[45] Christian Macho, Shane McIntosh, and Martin Pinzger. 2016. Predicting Build Co-Changes with Source Code Change and Commit Categories. In *Proc. of the International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 541–551.

[46] C. Macho, S. McIntosh, and M. Pinzger. 2018. Automatically repairing dependency-related build breakage. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 106–117.

[47] Shane Mcintosh, Bram Adams, and Ahmed E. Hassan. 2012. The Evolution of Java Build Systems. *Empirical Softw. Engg.* 17, 4-5 (Aug. 2012), 578–608. https://doi.org/10.1007/s10664-011-9169-5

[48] Shane McIntosh, Bram Adams, Meiyappan Nagappan, and Ahmed E. Hassan. 2016. Identifying and Understanding Header File Hotspots in C/C++ Build Processes. *Automated Software Engineering* 23, 4 (2016), 619–647.

[49] Shane McIntosh, Meiyappan Nagappan, Bram Adams, Audris Mockus, and Ahmed E. Hassan. 2015. A Large-Scale Empirical Study of the Relationship between Build Technology and Build Maintenance. *Empirical Software Engineering* 20, 6 (2015), 1587–1633.

[50] Xiangxin Meng, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2022. Improving fault localization and program repair with deep semantic features and transferred knowledge. In *Proceedings of the 44th International Conference on Software Engineering*. 1169–1180.

[51] Shaikh Mostafa, Xiaoyin Wang, and Tao Xie. 2017. PerfRanker: Prioritization of Performance Regression Tests for Collection-Intensive Software. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 23–34. https://doi.org/10.1145/3092703.3092725

[52] Eugene W. Myers. 1986. An O(ND) Difference Algorithm and Its Variations. *Algorithmica* 1 (1986), 251–266.

[53] Sangmin Park, Richard W. Vuduc, and Mary Jean Harrold. 2010. Falcon: Fault Localization in Concurrent Programs. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1* (Cape Town, South Africa) *(ICSE '10)*. ACM, New York, NY, USA, 245–254. https://doi.org/10.1145/1806799.1806838

[54] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The Strength of Random Search on Automated Program Repair. In *Proceedings of the 36th International Conference on Software Engineering* (Hyderabad, India) *(ICSE 2014)*. ACM, New York, NY, USA, 254–265. https://doi.org/10.1145/2568225.2568254

[55] Mohammad Masudur Rahman and Chanchal K. Roy. 2018. Improving IR-based Bug Localization with Context-aware Query Reformulation. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. ACM, New York, NY, USA, 621–632. https://doi.org/10.1145/3236024.3236065

[56] T. Rausch, W. Hummer, P. Leitner, and S. Schulte. 2017. An Empirical Analysis of Build Failures in the Continuous Integration Workflows of Java-Based Open-Source Software. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. 345–355.

[57] Zhilei Ren, He Jiang, Jifeng Xuan, and Zijiang Yang. 2018. Automated Localization for Unreproducible Builds. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 71–81. https://doi.org/10.1145/3180155.3180224

[58] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry. 2013. Improving bug localization using structured information retrieval. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 345–355. https://doi.org/10.1109/ASE.2013.6693093

[59] Mark Santolucito, Jialu Zhang, Ennan Zhai, Jürgen Cito, and Ruzica Piskac. 2022. Learning CI Configuration Correctness for Early Build Feedback. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 1006–1017. https://doi.org/10.1109/SANER53432.2022.00118

[60] Qusay Idrees Sarhan, Attila Szatmári, Rajmond Tóth, and Arpad Beszedes. 2021. CharmFL: A fault localization tool for Python. In *2021 IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 114–119.

[61] Tushar Sharma, Marios Fragkoulis, and Diomidis Spinellis. 2016. Does Your Configuration Code Smell?. In *Proceedings of the 13th International Conference on Mining Software Repositories* (Austin, Texas) *(MSR '16)*. ACM, New York, NY, USA, 189–200. https://doi.org/10.1145/2901739.2901761

[62] Matúš Sulír and Jaroslav Porubän. 2016. A Quantitative Study of Java Software Buildability. In *Proceedings of the 7th International Workshop on Evaluation and Usability of Programming Languages and Tools* (Amsterdam, Netherlands) *(PLATEAU 2016)*. ACM, New York, NY, USA, 17–25. https://doi.org/10.1145/3001878.3001882

[63] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. N. Nguyen. 2012. SYMake: a build code analysis and refactoring tool for makefiles. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 366–369. https://doi.org/10.1145/2351676.2351749

[64] C. Tantithamthavorn, R. Teekavanich, A. Ihara, and K. Matsumoto. 2013. Mining A change history to quickly identify bug locations : A case study of the Eclipse project. In *2013 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 108–113.

[65] Mohsen Vakilian, Raluca Sauciuc, J. David Morgenthaler, and Vahab Mirrokni. 2015. Automated Decomposition of Build Targets. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1* (Florence, Italy) *(ICSE '15)*. IEEE Press, Piscataway, NJ, USA, 123–133. http://dl.acm.org/citation.cfm?id=2818754.2818772

[66] Rijnard van Tonder and Claire Le Goues. 2018. Static Automated Program Repair for Heap Properties. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 151–162. https://doi.org/10.1145/3180155.3180250

[67] Carmine Vassallo, Sebastian Proksch, Anna Jancso, Harald C. Gall, and Massimiliano Di Penta. 2020. Configuration Smells in Continuous Delivery Pipelines: A Linter and a Six-Month Study on GitLab *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 327–337. https://doi.org/10.1145/3368089.3409709

[68] C. Vassallo, G. Schermann, F. Zampetti, D. Romano, P. Leitner, A. Zaidman, M. Di Penta, and S. Panichella. 2017. A Tale of CI Build Failures: An Open Source and a Financial Organization Perspective. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 183–193.

[69] Qianqian Wang, Chris Parnin, and Alessandro Orso. 2015. Evaluating the Usefulness of IR-based Fault Localization Techniques. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis* (Baltimore, MD, USA) *(ISSTA 2015)*. ACM, New York, NY, USA, 1–11. https://doi.org/10.1145/2771783.2771797

[70] Shaowei Wang and David Lo. 2014. Version History, Similar Report, and Structure: Putting Them Together for Improved Bug Localization. In *Proceedings of the 22nd International Conference on Program Comprehension* (Hyderabad, India) *(ICPC 2014)*. Association for Computing Machinery, New York, NY, USA, 53–63. https://doi.org/10.1145/2597008.2597148

[71] Shaowei Wang and David Lo. 2016. AmaLgam+: Composing Rich Information Sources for Accurate Bug Localization: Composing Rich Information Sources for Accurate Bug Localization. *Journal of Software: Evolution and Process* 28 (10 2016). https://doi.org/10.1002/smr.1801

[72] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware Patch Generation for Better Automated Program Repair. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. ACM, New York, NY, USA, 1–11. https://doi.org/10.1145/3180155.3180233

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

[74] R. Wen, D. Gilbert, M. G. Roche, and S. McIntosh. 2018. BLIMP Tracer: Integrating Build Impact Analysis with Code Review. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 685–694.

[75] David Widder, Michael Hilton, Christian Kästner, and Bogdan Vasilescu. 2018. I'm Leaving You, Travis: A Continuous Integration Breakup Story. In *International Conference on Mining Software Repositories (MSR)*. ACM, 165–169. https://doi.org/10.1145/3196398.3196422

[76] C. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei. 2014. Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 181–190. https://doi.org/10.1109/ICSME.2014.40

[77] Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, and Sunghun Kim. 2014. CrashLocator: Locating Crashing Faults Based on Crash Stacks. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis* (San Jose, CA, USA) *(ISSTA 2014)*. ACM, New York, NY, USA, 204–214. https://doi.org/10.1145/2610384.2610386

[78] X. Xia, D. Lo, S. McIntosh, E. Shihab, and A. E. Hassan. 2015. Cross-project build co-change prediction. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. 311–320. https://doi.org/10.1109/SANER.2015.7081841

[79] J. Xuan and M. Monperrus. 2014. Learning to Combine Multiple Ranking Metrics for Fault Localization. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 191–200. https://doi.org/10.1109/ICSME.2014.41

[80] Jifeng Xuan and Martin Monperrus. 2014. Test Case Purification for Improving Fault Localization. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Hong Kong, China) *(FSE 2014)*. ACM, New York, NY, USA, 52–63. https://doi.org/10.1145/2635868.2635906

[81] Xin Ye, Razvan Bunescu, and Chang Liu. 2014. 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* (Hong Kong, China) *(FSE 2014)*. ACM, New York, NY, USA, 689–699. https://doi.org/10.1145/2635868.2635874

[82] Fiorella Zampetti, Carmine Vassallo, Sebastiano Panichella, Gerardo Canfora, Harald Gall, and Massimiliano Di Penta. 2020. An empirical characterization of bad practices in continuous integration. *Empirical Software Engineering* 25, 2 (2020), 1095–1135.

[83] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk. 2006. On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering* 32, 4 (April 2006), 240–253. https://doi.org/10.1109/TSE.2006.38

[84] J. Zhou, H. Zhang, and D. Lo. 2012. 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)*. 14–24. https://doi.org/10.1109/ICSE.2012.6227210

[85] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang. 2019. An Empirical Study of Fault Localization Families and Their Combinations. *IEEE Transactions on Software Engineering* (2019), 1–1.